


QL COMPUTING

The background of the cover is a vibrant orange with stylized yellow and white lightning bolts. A large, three-dimensional rectangular prism is depicted, its top face colored orange and its side faces colored blue. The surfaces of the prism are covered in a fine grid pattern. Several jagged, stylized lightning bolts in red and green are shown striking the prism. A bright white starburst of light emanates from the top right corner of the prism. The overall aesthetic is that of a retro science fiction or technology magazine.

**IAN
SINCLAIR**

QL Computing



Digitized by the Internet Archive
in 2023

QL Computing

Ian Sinclair

GRANADA

London Toronto Sydney New York

Granada Technical Books
Granada Publishing Ltd
8 Grafton Street, London W1X 3LA

First published in Great Britain by
Granada Publishing 1984

Distributed in the United States of America
by Sheridan House, Inc.

Copyright © Ian Sinclair 1984

British Library Cataloguing in Publication Data

Sinclair, Ian, R.

QL computing

I. Sinclair QL (Computers)—Programming

I. Title

001.64'2 QA76.S625

ISBN 0-246-12595-0

Typeset by V & M Graphics Ltd, Aylesbury, Bucks
Printed and bound in Great Britain by
Mackays of Chatham, Kent

All rights reserved. No part of this publication may
be reproduced, stored in a retrieval system or
transmitted, in any form, or by any means, electronic,
mechanical, photocopying, recording or otherwise,
without the prior permission of the publishers.

Contents

<i>Preface</i>	vii
1 Setting Up The QL	1
2 Putting It All On The Screen	20
3 A Feast of Variables	33
4 Repeating Yourself	50
5 Strings and Other Things	67
6 Further Procedures	85
7 Channels, Microdrives and Data Filing	104
8 Windows and Other Effects	129
9 Guidance with Graphics	144
10 Sound Sense	158
<i>Appendix A: Editing</i>	169
<i>Appendix B: Using Printers</i>	171
<i>Index</i>	173

Preface

The QL computer offers much more memory for the user than any previous machine at a comparable price. Because of this, many buyers of the machine will be home computer users. Though the programs that are given away with the machine suggest that the design aim was business use, the feature of the Microdrive is more likely to attract the home user than the business user. The free programs are just as likely to be attractive to the home user, but unless the machine is correctly set up, and its operation understood, the buyer will not be able to get the best out of these programs. This book aims to offer just that type of help to the buyer.

In addition there will invariably be applications for your QL which cannot be covered by these programs. When your computing has moved to the stage where you need to design your own programs to solve your own problems, you will need to learn how to program the QL for yourself. Programming, after all, is one of the features for which a computer is intended. Owning a computer and not programming it for yourself is like buying a Porsche and getting someone else to drive it. If you have never programmed a computer for yourself, this book will show you how. If you have some experience of earlier Sinclair machines, such as the ZX machines, then this book will open a new world of programming to you. The QL does not use the same version of the BASIC programming language as the ZX machines, and it needs to be learned almost from scratch if you have previously used one of those microcomputers. This is also true if your previous experience has been with one of the other low-priced machines.

I would like to emphasise that this book was written while I was *using* a QL which had been delivered to my home, and that the listings of programs in this book were obtained from a printer that was connected to the QL. This might seem to be an unnecessary claim, but a few books have appeared which were written before any QL computers were available. Every program which appears in this book, and every

example of programming commands, has been tested on the QL which I have here in front of me. Nothing has been copied untested from a provisional manual, and where a command has not operated in the way that the new manual suggests, I have pointed out the difference. All of the early QL models appeared with some of the 'works' in the form of a separate plug-in box. There may be some differences when the final versions appear, but I do not expect these differences to affect the topics in this book. The final version may allow more commands to be carried out, but it will certainly not remove any of the existing ones. All of the screen displays which I have described were obtained on a TV receiver rather than on a colour monitor. This has been done because I feel that most of the readers of this book are likely to be using a TV receiver.

As always, I am greatly indebted to many people who made this book possible. Sinclair Research provided the machine, and Richard Miles of Granada Technical Books worked tirelessly to ensure that I had the QL on my desk as soon as possible. Among many others at Granada Technical Books, Sue Moore worked wonders with my manuscript, and the most efficient team of typesetters and printers that I know operated to produce the book in record-breaking time. I am sure that the result of all this work will be a book that will match the capabilities of your QL. I should add, incidentally, that I have no connection of any type with Sinclair Research or with Sir Clive Sinclair, other than the coincidence of name.

Ian Sinclair

Chapter One

Setting Up The QL

Your QL computer comes exceptionally well packed, and the package contains the QL itself, its power supply, and the TV cable, network cable, serial printer cable (included in the early versions), along with a large manual. The manual which originally came with my QL contained practically no instructions for helping the beginner, but the remainder of the manual arrived later. Unless you have a lot of experience with computers, you will probably need some more advice than the manual (which is very much better than most computer manuals) can spare space for. The power supply is the large black cube with the two sets of cables. One of these cables is connected permanently to a plug which has three miniature socket-holes in it. This is the plug that fits into the QL keyboard at the rear right-hand side (looking from the front). The socket for this plug is labelled 'POWER', and the plug should fit only one way round. Make sure that you are holding the plug correctly, with the bevelled portion uppermost, and don't on any account use force; it should be an easy (though not sloppy) fit. If you apply a lot of force, it is possible to insert the plug upside-down, and this could cause damage. The power supply cube is quite heavy, and you should be careful not to lift the QL without lifting or disconnecting the power supply as well. The other cable from the power supply must be fitted with a standard three-pin plug before you can start to make use of your QL.

The plug is connected as indicated in Fig. 1.1. There are only two leads, one blue and the other brown, and the cable should be tightly clamped. The fuse should be a 3 amp type, not the 13 amp variety which usually comes with the three-pin plug. If you are accustomed to fitting plugs for yourself then the diagram should be enough to remind you of what is needed. If you don't want to have anything to do with mains supplies, then take the power supply along to an electrician and get a plug, with a 3A fuse, connected. You don't have to take the whole computer, only the power supply box. When you have done this, make

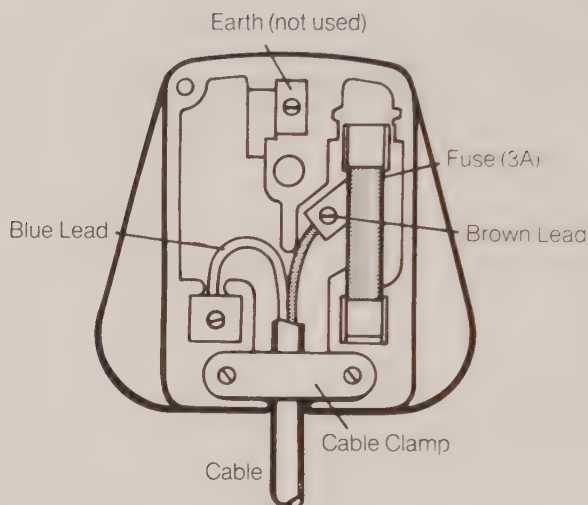


Fig. 1.1. The connections to a three-pin mains plug. Only the live and neutral leads are used. If you haven't done this sort of thing before, play safe and hand it to an electrician.

sure that the power supply box is located away from the QL and clear of the TV. Put it where the air can circulate round it. It gets only slightly warm while you are using the computer, but you should always try to keep it in a cool place. You may also want to place it on a felt pad or a polystyrene block, because if it rests on a table, it gives out an irritating humming or buzzing noise.

With that hurdle over, you are almost ready to put the QL to work for you, but you need the use of a TV receiver or a monitor. A computer is a device which is arranged so as to send out electrical signals that can be used to form images on a TV screen. There are two ways in which this can be done. One is to use a special form of TV display, which is designed to use the signals from the computer directly. Such a device is called a 'monitor', and it can't normally be used for receiving TV pictures from an aerial. The alternative method is to convert the signals from the computer into the same form as the signals that TV transmitters send, so that they can be connected to the aerial socket of an ordinary TV receiver.

There is an important difference between the two. Though it's convenient to be able to use a TV receiver (and cheap as well), the picture that you see is of a poor standard. This is because a TV receiver was never designed to accept computer signals, and also because of the need to convert the computer's signals into the same form as transmitted signals. The result is that the letters and other shapes on the

screen look fuzzy, and the colours look streaky. If you use a monitor, connected to the QL with a suitable cable, the shapes on the screen will be clearer, and the colours much better. This is a connection which your local computer store will have to help you with, because the Sinclair monitor cable (an extra) comes with only one plug, the one at the QL end. The computer store will have to fit the other plug which fits into the monitor. At the time of writing this book, I could not obtain suitable connectors, and I used a Fidelity MTV100, which is a combined TV and monitor, but with the TV input only.

Seeing is believing

Unless you connect a TV receiver or monitor to the QL you won't be able to see what the computer is doing. It will still compute for you just as well, but you won't see what is going on. To connect the QL to a TV receiver, you will need to plug the aerial lead into both the QL and the TV. Unless you can keep a TV receiver specially for use with the QL, you will find that you have to keep plugging and unplugging the aerial cable and the QL cable. This is never a good thing to have to do, because it wears out the socket on the TV, and a useful alternative is to use the type of adaptor that is illustrated in Fig. 1.2. This allows you to plug a lead into the aerial socket of the TV so that the TV can be used both for QL and for *Dynasty* without having to pull plugs out. The two-way TV adaptor that I used is sold in TV stores under the name of 'Panda'.

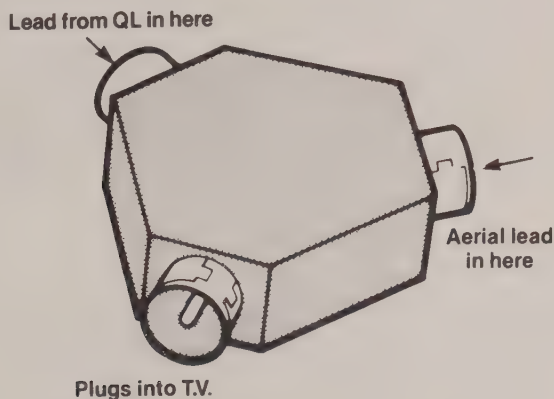


Fig. 1.2. A typical two-way adaptor for TV cables which you can buy at electrical stores.

4 QL Computing

You need to connect the QL to the TV or to the adaptor using the special cable that is provided with the QL. There are two different plugs on this cable and Fig. 1.3 shows the difference between these plugs. If you have used the adaptor, then all that you have to do to change between computing and TV watching is to switch channels!

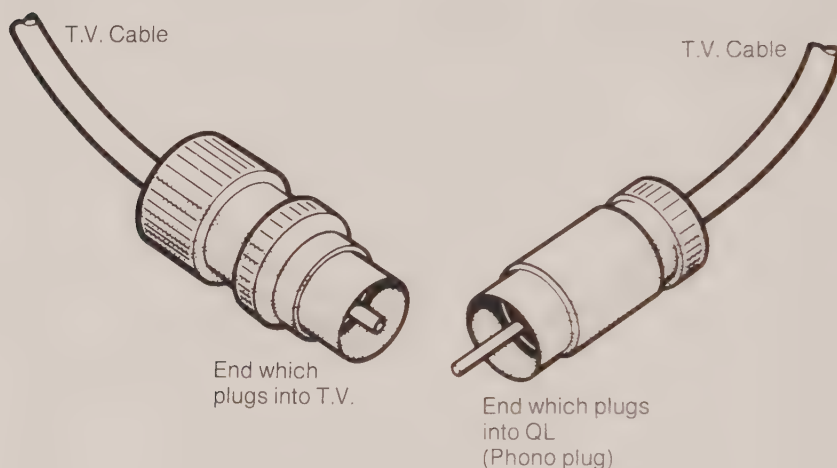


Fig. 1.3. The two different plugs on the TV cable.

The TV or monitor that you use to display the QL's signals need not be a colour receiver, not to start with at least. The skills of programming a QL do not require you to see the results in colour until you come to the colour instructions of the QL in Chapter 8. When you use a black-and-white TV or monitor to show the QL signals, the colours appear as shades of grey, and they are quite distinct. If you use a colour receiver, you will see the colours appear in all their glory, though not all makes of TV receivers will give equally good displays.

The big switch-on

Now before you plug in everything in sight and switch on, it's a good idea to see how many mains sockets you have around, and where you are going to house everything. When you are in full control of your QL you will need two mains sockets, one for the QL and one for the TV receiver. Most houses have desperately few sockets fitted, so you will find it worthwhile to buy or make up an extension lead that consists of

a three- or four-way socket strip with a cable and a plug (Fig. 1.4). This avoids a lot of clutter – you don't want to bring your QL crashing to the floor when you trip over a cable. Don't rely on the old fashioned type of three-way adaptor – they never produce really reliable contacts. The QL has no on/off switch, and you should *always* take out the mains plug after you have finished a computing session. The noise from the power-pack should be a useful reminder!

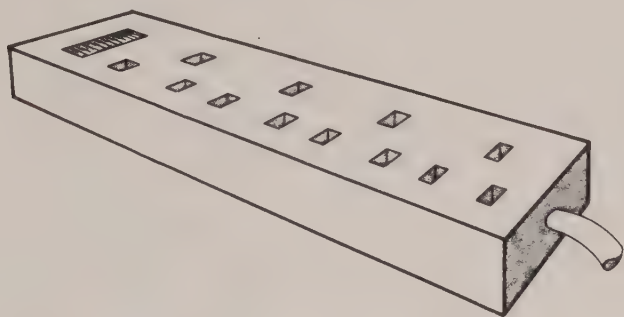


Fig. 1.4. A four-way socket strip which avoids the use of the old-style adaptors.

When you have the essential equipment to start computing, consisting of the QL keyboard, power supply, and TV (or monitor), quite a lot of flat surface is needed. Later on, you will probably want to add a printer and possibly disk drives and other extras which make the difference between having a computer *system* and just having a computer. All of this needs space, and the best way that I have found of organising this is one of the computer stands made by Selmor (Fig. 1.5). If you aren't at that stage yet, then a good-sized desk or table will have to suffice for the time being. Computing is like hi-fi – there's always something else that you can buy!

With everything housed and connected up, you now have to get used to some do's and don'ts. Unlike the low-price home computers which make use of cassette recorders for storing information, the QL uses a special tape system called the *Microdrive*. This uses miniature tape cartridges which are expensive and rather fragile. These cartridges come in a protective casing, and are used by plugging them into one of the horizontal slots at the front right-hand side of the QL. Don't attempt to use these cartridges until you have switched on the machine, and have some confidence in using it. In particular, *never switch the machine on or off when a cartridge is plugged in*. This can cause loss of

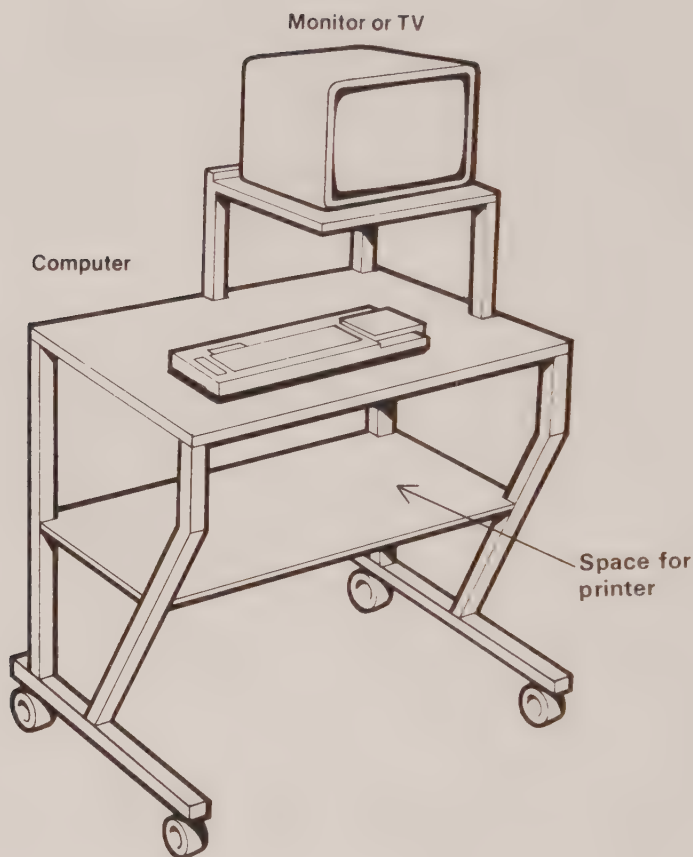


Fig. 1.5. How to house your QL and its auxiliary equipment. A typical computer stand made by Selmor.

information on the cartridge, and what you lose might just be the program that you most wanted to use. In fairness, I must add that, in the course of testing, I never found that any harm was done to any of my Microdrive cartridges in this way. In addition to this 'don't', you have another thing to attend to if your QL is one of the early models. These QLs (mine is one of them) came with a small box which had to be plugged into the socket that is marked 'ROM'. This socket is at the rear left-hand side of the computer, and the box is fitted with the label upwards. On the early models, *this box must be in place when the computer is switched on*. You must not attempt to use the computer without this box in place, because the box contains the essential 'works' of the computer. Later deliveries of the QL will have the 'works' totally inside, so that the box will not be needed. Eventually, all the boxes will

be exchanged for internal fittings, though at the time of writing, Sinclair Research has not indicated how this will be done.

The next step, then, is to switch on the TV receiver and the QL. Unless you are exceptionally lucky, or using a monitor, you will probably see nothing appear on the TV screen. This is because a TV receiver has to be tuned to the signal from the QL. Unless you have been using a video cassette recorder, and the TV has a tuning button that is marked 'VCR' it's unlikely that you will be able to get the QL tuning signal to appear on the screen of the TV simply by pressing tuning buttons. The next step, then, is to tune the TV to the QL's signals.

Figure 1.6 shows the three main methods that are used for tuning TV receivers in this country. The simplest type is the dial tuning system that is illustrated in Fig. 1.6(a). This is the type of tuning system that you find on black/white portables, and to get the QL's signal on the screen, you only have to turn the dial. If the dial is marked with numbers, then you should look for the signal somewhere between numbers 30 and 40. If the dial isn't marked, which is unusual, then start with the dial turned fully anticlockwise as far as it will go, and slowly turn it clockwise until you see the QL signal appear.

What you are looking for, if the QL hasn't been touched since you switched it on, is the screen display that is illustrated in Fig. 1.7. The bottom line of this display is a copyright notice – on my QL it carried the date of 1983. Above it is the box which carries the monitor/TV instructions. The reason for these is that if you use a TV, the QL will have to place bigger letters on the screen. Pressing the key that is marked 'F2' will arrange this, giving you a screen display that is easily visible. One of the Microdrives will switch on briefly when you press the F2 (or F1) key, but if there is no cartridge in the drive, there will be no delay. If you are using a TV, and you press the 'F1' key for monitor use, you will see the screen divide into three sections (white, red and black), and you may not be able to see any typed letters at the left-hand side. If you find that you have selected the wrong type of display, you can reselect. Make certain that you have *no Microdrive cartridge inserted*, then press the RESET button at the right-hand side of the keyboard. The screen will display a patchwork pattern for a moment, and then the message that is illustrated in Fig. 1.7 will reappear.

When you can see the words as in Fig. 1.7, turn the dial carefully, turning slightly in each direction until you find a setting in which the words are really clear. On a colour TV receiver the words may never be particularly clear, but get them steady at least and as clear as possible.

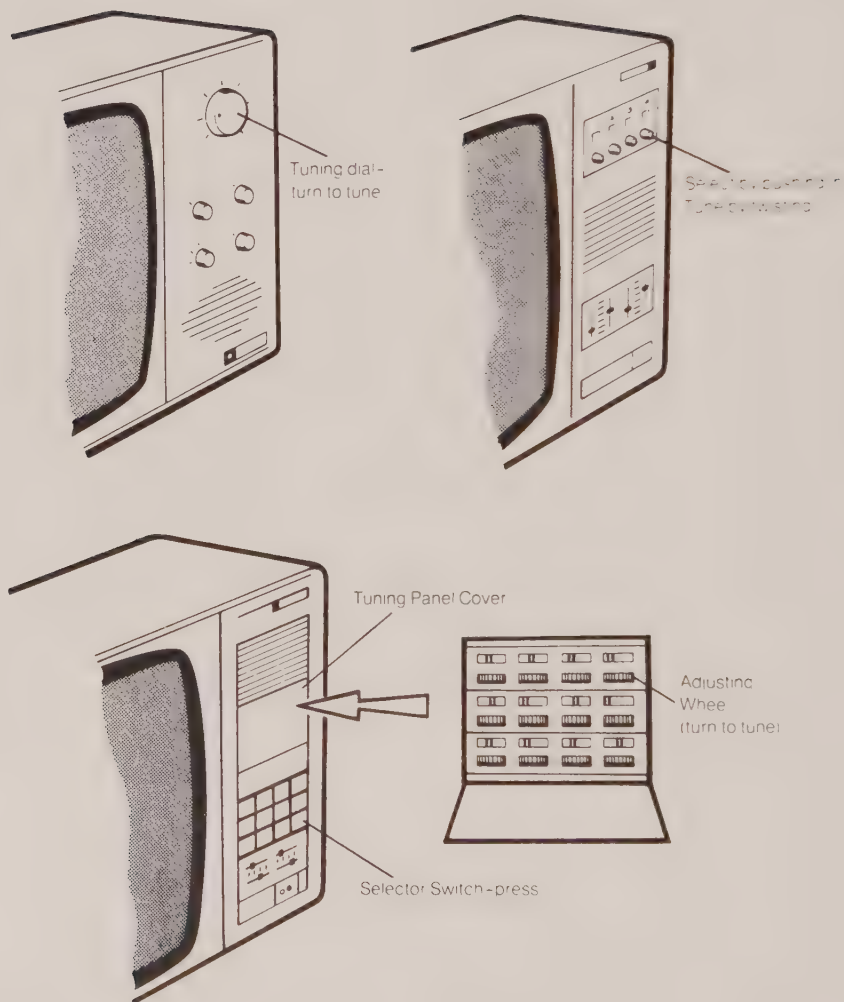


Fig. 1.6. TV tuning controls. (a) Single dial, as used on black and white portables; (b) four-button type; (c) the more modern touch-pad or miniature switch type.



Fig. 1.7. The message on the screen at switch-on.

The older types of colour and B/W TV receivers used mechanical push-buttons (Fig. 1.6(b)) which engage with a loud clonk when you push them. There are usually four of these buttons, and you'll need to use a spare one which for most of us means the fourth one. Push this one fully in. Tuning is now carried out by rotating this button. Try rotating anticlockwise first of all, and don't be surprised by how many times you can turn the button before it comes to a stop. If you tune to the QL's signal during this time, you'll see the message on the screen. If you've turned the button all the way anticlockwise and not seen the tuning signal, then you'll have to turn it in the opposite direction, clockwise, until you do. If you can't find the QL signal at any setting, check that you have connected the cables to the TV aerial socket correctly. If you are using the Panda adaptor, you can push one of the other tuning buttons to check that you can receive normal TV signals. If you can, there's nothing wrong with the TV, so switch back and try again to find the QL signal.

Modern TV receivers are equipped with touch pads or very small push-buttons for selecting transmissions. These are used for selection only, not for tuning. The tuning is carried out by a set of miniature knobs or wheels that are located behind a panel which may be at the side or at the front of the receiver (Fig. 1.6(c)). The buttons or touch pads are usually numbered, and corresponding numbers are marked on the tuning wheels or knobs. Use the highest number available (usually 6 or 12), press the pad or button for this number, and then find the knob or wheel which also carries this number. Tuning is carried out by turning this knob or wheel. Once again, you are looking for a clear picture on the screen.

On this type of receiver, the picture is usually 'fine-tuned' automatically when you put the cover back on the tuning panel, so don't leave it off. If you do, the receiver's circuits that keep it in tune can't operate, and you will find that the tuning alters, so that you have to keep re-tuning. The QL should give a good picture on practically any TV receiver. I tried it with several, and even my Philips portable colour TV, which does not work well with computers, gave a reasonably good picture with the QL. If your TV exhibits faults like a shaking picture, or very blurred colours, then check the tuning carefully. If the faults persist, and the TV is correctly tuned, you will have to contact the service agents for the TV - or use a different model in future!

And now to work ...

Once you have achieved a tuned signal from your QL, the business of mastering the use of the QL begins. Once you have pressed key F1 or F2 to select your display, you'll see a flashing (blue) square near the bottom of the screen. This is called the *cursor*. It is used as a marker, and when you press a key, a letter (or number, or whatever is marked on the key) will appear at the position of the cursor. The cursor will then move across the screen to the next position. It's important to note at this point that nothing that you can do just by pressing keys on the keyboard can possibly damage the QL machine - the worst you can do is to lose a program that was stored in the memory. You can, however, damage the QL by spilling coffee all over it, dropping it, or connecting it up to other circuits while the power is switched on. You can also damage Microdrive cartridges if you attempt to take them out while they are still running. *Always* switch off the computer, and everything that is connected to it when you insert or remove any of the plugs at the back. *Never* switch off the computer, or remove a Microdrive cartridge while the cartridge is being used. You can tell when a Microdrive is in use because you can hear the tiny electric motor driving the tape around, and you can see the indicator light at the left-hand side of the Microdrive slot. The numbers of the Microdrives are not printed on the machine. The left-hand one is No. 1 (in commands, MDV1), and the right-hand one is No. 2 (MDV2). We'll look at how to make use of these Microdrives in a moment.

Key-tapping time

It's time now to look at the keyboard, because the keyboard is the way that you pass instructions to the QL. If we ignore the keys F1 to F5 at the left-hand side, then most of the QL keys look like typewriter keys. The arrangement of letters and numbers is the same as that of a typewriter and if you've used a typewriter, particularly an electric typewriter, then you should be able to find your way round the keyboard of the QL pretty quickly. When you press any of the letter keys, you will see the letter appear on the screen. What you see is a *lower-case* (i.e. small) letter, not an *upper-case* (i.e. capital) letter. Just like a typewriter, the keyboard of the QL normally gives you lower-case letters. If you want a capital, you need to press one of the two SHIFT keys as well as a letter key. Commands that you give to the computer in typed form can be in either lower-case or upper-case. If you want capitals all the time, press the key that is marked CAPS LOCK. To return to the original system, you only have to press the CAPS LOCK key again. Unfortunately, there is no indicator light to show you whether CAPS LOCK is on or off – you just have to try it and see! Whatever the setting of the CAPS LOCK key happens to be, the keys which show two symbols on them, like most of the number keys, will need the use of SHIFT to get the symbol on top. When you press one of these keys alone, you get the symbol that is marked on the lower part of the key. Using SHIFT along with the key gives the symbol on the upper part of the key. For example, if you press the '8' key by itself, you get 8. If you press the '8' key and SHIFT together, you get '*'.

As well as the ordinary typewriter keys, there are a number of special keys which are not found on any typewriter. At the left-hand side of the keyboard, for example, you will find the keys that are labelled ESC (escape) and CTRL (control). These ESC and CTRL keys are used in ways that are outlined in the manual – the names aren't really a good description of what they do. They are of more use in the programs that accompany the QL than for the programs that you are likely to write for yourself. One action that you should know about at the moment, however, is the letter delete. Type a word, and then press CTRL and the left-arrow key (next to CTRL). You'll see that one letter disappears off the end of the word that you have typed. If you hold both keys down, you'll see the other letters being deleted also, one by one, rapidly, as the cursor moves left. This is one of the useful ways that you can rub out mistakes in your typing. The CTRL key and the spacebar, pressed together, will stop a program from running. When you stop a program in this way, you can start it again, because the program is *not* wiped

from the memory. Another way to stop a program from running is to use the RESET button at the right-hand side, but this always has the effect of wiping out any program from the memory. We'll come back to that important point later.

The set of four keys that are marked with arrows are 'cursor' keys. The more useful two are at the left of the long 'spacebar', and the other two are at the right. The two at the left are used to correct mistakes in your typing, so that you can rub out a letter, replace one letter with another, or insert letters where you want. That's something else that we'll come back to later, in Appendix A. We have already briefly looked at the effect of the left-arrow key when it is pressed at the same time as the CTRL key. Finally, the set of five keys at the left-hand side of the keyboard comprises what are called the 'programmed function' keys. They are used in the programs which are packed with your QL.

The most important of all of the special keys, however, as far as we are concerned at the moment, is the key that is marked ENTER. This is in the position of the 'carriage return' key of an electric typewriter, but its action is not the same in all respects. Pressing the ENTER key is a signal to the computer that you have completed typing an instruction and that you now want the computer to obey it. If you are accustomed to using an electric typewriter, you will have to change some of your habits as far as this key is concerned. During the use of a typewriter, you would press the 'carriage return' key each time you wanted to select a new line, with typing starting at the left-hand side of the new line. The ENTER key of the computer does rather more than this. If the material that you are typing into the QL takes more than one line on the screen, the machine will *automatically* select the next screen line for you. The ENTER key must *not* be used for this purpose. The ENTER key is used only when you want the machine to carry out a command or store an instruction, not simply when you want to use a new line. It will always provide a new line for you, however, and select a position at the left-hand side. The position where a letter or other character will appear when you press a key is indicated by the cursor.

You will find that the action of each key repeats if you hold your finger on it, and this repeat action is quite fast. If you type a set of meaningless letters and then press ENTER, the computer usually responds with the phrase:

bad name

This is because the computer is a simple machine. It can understand only a few words, the words that we call its 'reserved words' or 'instruction words'. If what you type does not include these words, or

uses these words incorrectly, then this is a 'bad name' as far as the computer is concerned. It may make sense to you, but it doesn't make sense to the computer!

Microdrive tryout

The QL computer, like all others, stores instructions and other information in its memory – but only while the machine is switched on. Whenever you switch off your QL, anything that was stored in its memory is instantly lost, and you can't get it back even if you switch on again very quickly. For that reason, we need to store the programs and the information that we need to use in programs in another form. The most useful forms are magnetic disks and magnetic tape. The QL has at the time of writing no 'official' way of using magnetic disks, such as are used for business applications, but uses the Microdrive instead. Several 'unofficial' ways of connecting the QL to disk systems have, however, been advertised.

The computer has circuits which will convert the instructions of a program into signals, which can then be recorded on the tape of the Microdrive. When these signals are replayed, another set of circuits will convert the signals back into the form of a program. In this way, the use of the QL Microdrive allows you to record your programs on tape and to replay them again. Before you tackle the rest of this book, then, it's important to check now that you can use the Microdrive to record and replay programs.

The easiest thing to try first is to use the Microdrive to put a program into memory. This is called 'loading', and since your QL comes with a set of four programs on Microdrive cartridges, these are a convenient way of getting experience. You will see from the notes that come with the programs that you are not supposed to make use of the programmed cartridges. This does not mean that the Microdrive is particularly unreliable, it is simply a normal safeguard. All recording systems are subject to loss of signals, and people who use computers always keep several copies of anything that is valuable. During the time of writing this book, two of my Microdrive cartridges became unusable. Your manual tells you how to make copies of these programs. This procedure is called 'making a backup', but until you have acquired some experience with the Microdrives, you may very easily make mistakes in the backup procedure which will cause you to lose a valuable program. Leave backing up until later, when you are more confident. Note, incidentally, that it's really rather foolish to pay a

lot of money for a program that you can't copy. If a program-seller announces proudly that his program can't be copied, then *don't buy it!*

Let's try to load the EASEL program, then. You should have the computer switched on, with nothing in its memory. If you are in any doubt, press the RESET button, then the correct F1 or F2 key, depending on whether you are using a monitor or a TV receiver. You will then be ready to load the program. Take the Microdrive cartridge out of its casing, and hold it by the ribbed ends with the name label uppermost. Now push it gently into the left-hand Microdrive slot, Microdrive No. 1. It should push in easily, with very little resistance. If it seems stiff, don't force it. Check that the cartridge is the right way round, and if you still find it hard to insert, try another program cartridge.

Simply inserting the cartridge does not cause it to operate, however. You must issue the correct commands to the computer first, and this is where you get your first introduction to computer awkwardness. A command to a computer must be exact. You have to type the command in exactly the right form, otherwise it will not be acted upon. It's like a slot-machine – it will not work with 'foreign coins'! One thing in particular that you have to watch is the underline mark (). This is on the key to the right of the zero, on the top line. You *must* hold down the SHIFT key when you use this, otherwise you get a normal dash (–). The loading commands for the Microdrive will not work with the dash, only the underline mark. The other point is that if you press the spacebar when you have the SHIFT key pressed, you get no space! This can cause mistyping, with words run together, and the QL will not accept such mistyped commands.

To load EASEL, then, you have to type, exactly as shown,

```
LRUN MDV1_BOOT
```

and then press the ENTER key. I know that the manual advises using the command EXEC_W MDV1_EASEL, but this did not work on my QL. You can type the command in lower-case (small) letters if you like, but you *must* use the underline mark correctly. If the command is correctly typed, then you will hear the Microdrive motor start, and you will see the Psion copyright notice appear. This doesn't mean that the program is loaded, just the title! You will have to wait for some time yet before you can use the program, and this is one of the significant differences between using Microdrives and using disks. Another is that blank disks cost only about £1.50 each! There is an alternative loading method for this and the other three free programs. This is to switch on the machine, then insert the Microdrive cartridge in the MDV1 slot

before you press F1 (or F2). When you press the F-key, the program will load and start automatically.

If you use the EXEC_W command that is mentioned in the manual, the Microdrive runs, and then the computer locks up. You will then have to remove the cartridge, and press the RESET button to restore normal service. Once you have loaded the EASEL program, you might like to look at the instructions in the manual about using it, and try it out for yourself. It will give you a lot of useful experience of operating a computer. It will also show you, if you are using a TV receiver, how hard it is to read words in colour on the screen of an ordinary TV. This is why monitors are always specified by business users.

Recording your own

Once you have gained some experience with the Microdrive, you might try the procedure that the manual suggests for backing up a program. Note that this introduces another example of the principle that you can make a Microdrive program load and run automatically by having the cartridge in the MDV1 position when you press the F1 or F2 key after resetting. This is an exception to the rule that you always RESET with the cartridge out. You *must* be very careful to have the cartridges in the correct slots. If you don't you *could* find that you have copied a blank cartridge on to a valuable program cartridge. This is why I have suggested leaving the backing up operation until you are more experienced with the Microdrives. The next step is to create your own program, record it, and then replay it. First of all, you need to *format* a blank cartridge.

Formatting is an operation that is rather like ruling lines on a sheet of blank paper. The Microdrive tape is an endless loop, with a join in it. You can't reliably record on the join, and the machine must place signals at intervals on the tape to mark the tape into pieces that can be used. This marking operation allows the computer to find information more easily, just as you find it easier to locate a word in a book if you know it's on page 33. A blank Microdrive tape cannot be used until this formatting operation has been carried out. Place the new cartridge into the MDV1 slot, and then type the command:

```
format mdv1_test
```

and then press ENTER. The name 'test' is a name that will be recorded on the cartridge; you can use any name that you like provided it has no more than ten letters. If you use a name of more than ten letters, the

extra letters will be ignored. You must not use punctuation marks, such as commas, semicolons or full-stops in the name. The only mark that you can use, in fact, is the underline. To check the name, and to see what has been recorded on a cartridge at any time, type:

```
dir mdv1_
```

and press ENTER. The underline bar at the end of MDV1 is very important, and the command will not operate if you forget it - you will just get the message 'not found'. Once you have a formatted Microdrive tape, you can make a recording on it to check the action.

Now before you can make a recording to test the system, you need a program to record, and this involves some typing. This is easy if you have just switched the QL on, but if you have been pressing keys at random, then it's a good idea to switch off again, then on, with any Microdrive cartridge removed.

Type the number 10 (1 and then 0), and then the word rem. It doesn't matter whether you type rem or REM. Check that this looks correct, and then press the ENTER key. The effect of this is to place the instruction line '10 REM' into the memory of the QL. As you type the first digit, the character will be seen on the screen at the cursor position. When you press the ENTER key, the cursor moves to the next line down. At the same time, your command appears at the top of the screen, in the form:

```
10 REMark
```

If you see a mistake as you type the line, just use the back-space action, by pressing the CTRL key and the left-arrow key together. This shifts the cursor backwards, and deletes the letter that the cursor is now over. You can then type the correct letter, which will replace the incorrect one. If this makes the line correct, pressing ENTER will enter it into the computer. If you have typed something that is incorrect, like REN or RWM then you will not be warned in any way - the computer accepts anything that you type following a number. To correct a line such as:

```
20 Ren
```

just type the correct version:

```
20 rem
```

and press the ENTER key. Note this is not like the action of the older ZX series of computers. Now type the rest of the lines, as illustrated in Fig. 1.8, remembering to press the ENTER key after you have completed typing each line. The numbers are called 'line numbers', and


```

10 REMark
20 REMark
30 REMark
40 REMark

```

Fig. 1.8. A program for testing the Microdrive recording and replaying actions.

they are present for two reasons. One is to remind the computer that this is a program, the other is to guide it, because the computer will normally carry out instructions in the same order as the line numbers. You can check that your program looks correct by asking the computer to *list* it. Listing means that the computer prints on the screen whatever you have stored in its memory. Using the 'line numbers' ensures that the instructions are stored, and if you type 'list', and then press ENTER, you will see your program. Don't be surprised to find that all the lower-case letters (like *rem*) have been converted to upper-case (like *REM*), because this is part of the action of the computer, along with filling in the rest of the word. Check from this 'listing' that the program is like the printed version in Fig. 1.8.

Now make sure that the Microdrive is ready, with the cartridge in place in position MDV1. Now type:

```
SAVE MDV1_mytest
```

and press the ENTER key. The word 'save' is the instruction to the computer meaning that you want to save (record) a program onto a cartridge, and the *mdv1* means that you are using the Microdrive No. 1 for this purpose. The 'mytest' part is a 'filename' which the computer will use to recognise the program if it is asked to. You must add this following the underline bar. It is possible to save a program without a filename, as long as you have the underline present, but you should not make a habit of this. Normally, you will keep a number of different programs on each cartridge, and you need the filenames to instruct the computer to load back the correct one. When you press the ENTER key, the motor of the Microdrive will run, and you will soon see the cursor reappear on the screen. Even though the Microdrive motor is still running, the program is recorded, and you can type *dir mdv1_* to check it (then press ENTER). The motor of the Microdrive will stop automatically at the end of the process. This lets you know that the program has been recorded. That's all that's involved in making the recording. Now comes the crunch. You have to be sure that the recording was OK. Type *NEW* and press ENTER. This should have wiped your program from the memory. Just to make it look better, type *CLS* and press ENTER. This clears the screen, so that there will be no

confusion. Now type LIST and press ENTER. Nothing should appear

LIST means put a list of the program instructions on the screen, and there shouldn't be any!

You can now load the instructions in from the tape. Type LOAD MDVI_mytest and then press ENTER. The Microdrive motor will run very briefly and then stop. Type LIST now, then press the ENTER key. You should see your program appear on the screen. Once you can reliably save programs on tape, and reload them, you can confidently start computing. When you have spent an hour or more typing a program on to the keyboard, it's good to know that a few minutes more work will save your effort on tape so that you won't have to type it again. QL Microdrive cartridges cost rather more than ordinary cassettes, but they work a lot better, and with no need for adjustments. The only thing that you have to be careful about is the care of your Microdrive cartridges. The table in Fig. 1.9 should act as a useful reminder to you in this respect.

The Microdrive tape is very narrow, and moves much faster than an ordinary cassette tape. It therefore jams easily, and when this happens, the Microdrive will switch off without loading or storing, and will eject the cartridge slightly. A jammed cartridge can *sometimes* be freed by slapping it against a hard surface, like a desk-top.

The main dangers to cartridges are contamination of the tape, which will cause a 'bad medium' error message. Take the following precautions:

- (1) Keep your fingers away from the tape. Hold the cartridge only by its ridged end.
 - (2) Never poke anything (ball pen, pencil) into the cartridge.
 - (3) Never turn the computer on or off with cartridges in place.
 - (4) Keep spare cartridges in their cases, and store in a cool dry place.
 - (5) Be careful how you insert and remove cartridges.
 - (6) Press the cartridge firmly into place before starting the Microdrive. Sometimes a cartridge may have to be held in place.
 - (7) Don't insert a cartridge unless you are going to run it.
-

Fig. 1.9. A note of some points about Microdrive cartridges.

Load and run

Later on, when you start to write your own programs, you will want to know how to make programs load and run automatically when the F-key is pressed. The method is quite simple. If you record a program

whose filename is **BOOT**, then this program will load and run automatically if it is on a cartridge that is in MDV1 when you press the F-key! The QL is organised so that MDV1 switches on when you press an F-key. If there is no cartridge in the drive, it switches off at once, but if there is a cartridge, the QL looks for a program called 'BOOT'. If there is one, it loads it and runs it. Your **BOOT** program can be a very simple one, just a command to load another program, or it can be the program which you want to use most often. You could also use it to load *and* run another program, by making the **BOOT** program contain an **LRUN** instruction (see the Manual for details when you have finished reading this book).

Chapter Two

Putting It All On The Screen

Chapter 1 will have broken you in to the idea that the QL, like practically all computers, takes its orders from you when you type them on the keyboard. You will also have found that an order is obeyed when the ENTER key is pressed. You will by now have used the command LIST which prints your program instructions on to the screen. There are two other useful points that you need to know before we go much further. One is that you can clear the screen by typing CLS (or cls), and then pressing the ENTER key. The instructions at the bottom of the screen are *not* erased in this way, however. As your familiarity with the computer keyboard increases, you will want to make use of the editing commands, and these are explained in Appendix A.

Now there are two ways in which you can use a computer. One way is called *direct mode*. Direct mode means that you type a command, press ENTER, and the command is carried out at once. This can be useful, but the more important way of using a computer is in what is called *program mode*. In program mode the computer is issued with a set of instructions, with a guide to the order in which they are to be carried out. A set of instructions like this is called a *program*. The difference is important, because the instructions of a program can be repeated as many times as you like with very little effort on your part. A direct command, by contrast, will be repeated only if you type the whole command again, and then press ENTER. The set of command words that can be used, along with the rules for using them, make up what is called a 'programming language'. The QL provides you with a new and very modern version of the most commonly used of all programming languages for small computers, BASIC. BASIC is short for 'Beginners' All-purpose Symbolic Instruction Code', and it was originally devised for teaching purposes. Since then, it has developed into a useful language in its own right. The version of BASIC that your QL uses is, however, enriched by a lot of extra commands.

An important point about all computer commands, whether they are

direct commands or program instructions, is that they have to be in a precise form. This is particularly true of the QL, and if you have programmed a ZX-81 or Spectrum previously, you will have to get rid of some old habits. You will have found already how the underline character () has to be used in the Microdrive commands, and as you go through this book you will discover the importance of spaces. Most of the command words of the QL have to be followed by a space. This is unusual, and if you have used almost any other computer, you may have become accustomed to leaving out spaces. The usual QL reaction to leaving out a space is the 'bad name' error message. The examples in this book will show you the few cases where you can leave out spaces, but it is always safer to assume that a space will be needed between a command word and whatever follows it.

Let's now take a look at the difference between a direct command and a program instruction. If you want the computer to carry out the direct command to add two numbers, 1.6 and 3.2, then you have to type:

PRINT 1.6 + 3.2 (and then press ENTER)

You have to start with PRINT (or print), because a computer is a dumb machine, and it obeys only a few set instructions. Unless you use the word PRINT, the computer has no way of telling that what you want is to see the answer on the screen. It doesn't recognise instructions like 'GIVE ME' or 'WHAT IS' – only a few words that we call its *reserved words* or *instruction words*. PRINT is one of these words. So that you can recognise these reserved words more easily in this book, I shall always print them in upper-case (capital) letters from now on. You know by now, however, that you can type them in either upper-case or lower-case. You will *always* see them in upper-case when you LIST your program, whether on the screen or on a printer. Remember that there *must* be a space following the 'T' of PRINT. You do not need to have spaces between the '6' and the '+' or the '+' and the '3'.

When you press the ENTER key after typing PRINT 1.6 + 3.2, the screen shows the answer, 4.8. This answer is not shown in the same place as your typed command, however. If you have just cleared the screen before typing, the answer will appear at the top left-hand corner. If there is anything else printed on this main part of the screen, the answer of 4.8 will appear on the next line. When you are working with a TV, the QL clears its screen to a red background colour. Results of program commands appear in white on this, but program listings appear in a white box, using red characters. The commands that you type appear as green letters on a black background near the bottom of

the screen. If you are using a monitor, the screen is divided differently. The letters that you type appear at the bottom as before, but the division between listings and results is done on to side-by-side boxes.

Once a direct command has been carried out, however, it's finished. A program does not work in the same way. A program is typed in, but the instructions of the program are not carried out when you press the ENTER key. Instead, the instructions are stored in the memory, ready to be carried out as and when you want. The computer needs some way of recognising the differences between your commands and your program instructions. On computers that use the 'language' called BASIC this is done by starting each program instruction with a number which is called a line number. This must be a positive whole number, the type of number that is called a positive integer. This is why you can't expect the computer to understand an instruction like $5.6 + 3 =$; it takes the 5 as being a line number, and the rest doesn't make sense.

Let's start programming, then, with the arithmetic actions of add, subtract, multiply and divide. Computers aren't used all that much for calculation, but it's useful to be able to carry out calculations now and again. Figure 2.1 shows a four-line program which will print some arithmetic results.

```
10 PRINT 5.6+6.8
20 PRINT 9.2-4.7
30 PRINT 3.3*3.9
40 PRINT 7.6/1.4
```

Fig. 2.1. A four-line arithmetic program.

Take a close look at this, because there's a lot to get used to in these four lines. To start with, the line numbers are 10,20,30,40 rather than 1,2,3,4. This is to allow space for second thoughts. If you decide that you want to have another instruction between line 10 and line 20, then you can type the line number 15, or 11 or 12 or any other whole number between 10 and 20, and follow it with your new instruction. Even though you have entered this line out of order, the computer will automatically place it in order between lines 10 and 20. If you number your lines 1,2,3 then there's no room for these second thoughts though you can change line numbers if you have to by using the editing commands.

The next thing to notice is how the number zero on the screen is slashed across. This is to distinguish it from the letter O. The computer simply won't accept the 0 in place of O, nor the O in place of 0, and the slashing makes this difference more obvious to you, so that you are less

likely to make mistakes. The zero that you see on the keyboard *isn't* slashed, but it is on a different key, and is differently shaped. Type some zeros and Os on the screen so that you can see the difference.

Now to more important points. The star or asterisk symbol in line 30 is the symbol that the QL uses as a multiply sign. Once again, we can't use the \times that you might normally use for writing multiplication because this is a letter. There's no divide sign on the keyboard either, so the QL, like all other small computers, uses the backslash (/) sign in its place. This is the diagonal line on the same key as the question mark, *not* the one on the key just above and to the right of the ENTER key.

So far, so good. The program is entered by typing it, just as you see it. You don't need to leave any space between the line number and the P of PRINT, because the QL will put one in for you when it displays the program on the screen. You *must* leave a space following PRINT though, and I have to emphasise this because few computers are quite so fussy. The QL is very intolerant about this sort of thing, so don't be surprised when you get 'bad name' messages. Getting back to the program example, you will have to press the ENTER key when you have completed each instruction line, before you type the next line number. You should end up with the program looking as it does in the illustration. When you have entered the program by typing it, it's stored in the memory of the computer in the form of a set of code numbers. These are two things that you need to know now. One is how to check that the program is actually in the memory; the other is how to make the machine carry out the instructions of the program.

The first part is dealt with using the command LIST that you know already. You can use the CLS command to wipe the screen first if you like, then type LIST and press the ENTER key. When you press the ENTER key, and not before, your program will be listed on the screen. You will then see how the computer has printed the items of the program on the screen, with spaces between the line numbers and the instructions. To make the program operate, you need another command, RUN. Type RUN, then press the ENTER key, and you will see the instructions carried out. To be more precise, you will see:

```
12.4
4.5
12.87
5.428571
```

That last line should give you some idea of how precisely the QL can carry out this type of arithmetic. A calculator will give several more places of decimals, but six places is often enough for practical purposes.

You'll notice, by the way, that the results are printed over the program listing. You can avoid this by using CLS (press ENTER) before you use RUN.

When you follow the instruction word PRINT with a piece of arithmetic like $2.8*4.4$, then what is printed when the program runs is the *result* of working out that piece of arithmetic. The program *doesn't* print $2.8*4.4$, just the result of the action $2.8*4.4$.

Now this is useful, but it's not always handy to get a set of answers on the screen, especially if you have forgotten what the questions were. The QL allows you a way of printing anything that you like on the screen, exactly as you type it, by the use of what is called a *string*.

Figure 2.2 illustrates this principle. In each line, some of the typing is enclosed between quotes (inverted commas) and some is not. The

```
10 PRINT "2+2=" ; 2+2
20 PRINT "2.5*3.5=" ; 2.5*3.5
30 PRINT "9.4-2.2=" ; 9.4-2.2
40 PRINT "27.6/2.2=" ; 27.6/2.2
```

Fig. 2.2 Using quote marks. Whatever you type between the quote marks is printed on the screen, but not the quote marks themselves.

semicolons are *very* important, and you must not omit them in the way that you can on a lot of other computers. Enter this short program, clear the screen, and run it. Can you see how very differently the computer has treated the instructions? Whatever was enclosed between quotes has been printed exactly as you typed it. Whatever was not between quotes is worked out, so that the first line, for example, gives the unsurprising result:

$2+2=4$

Now there's nothing automatic about this. If you type a new line:

```
15 PRINT "2+2=" ; 5*1.5
```

then you'll get the daft reply, when you RUN this, of:

$2+2= 7.5$

The computer does as it's told and that's what you told it to do. Only a looney would believe that computers could take over the world!

This is a good point also to take notice of something else. The line 15 that you added has been fitted into place between lines 10 and 20 – LIST if you don't believe it. No matter in what order you type the lines

of your program, the computer will sort them into order of ascending line number for you.

With all of this accumulated wisdom behind us, we can now start to look at some other printing actions. PRINT used alone in this way always means print on to the TV screen. For activating a paper printer ('hard copy', it's called), there's a separate variety of PRINT instruction which is followed by a number (such as PRINT #3,). These instructions are not useful to you unless you have a printer connected. Appendix B illustrates how to use a serial printer with the QL.

Now try the program in Fig. 2.3. You can try typing the lines in any order that you like, to establish the point that they will be in line-number order when you list the program. When you clear the screen

```
10 PRINT "This is"  
20 PRINT "the remarkable"  
30 PRINT "QL Computer."
```

Fig. 2.3. Using the PRINT instruction to place words on the screen. Each PRINT command causes a separate screen line to be selected.

and RUN the program, the words appear on three separate lines. This is because the instruction PRINT doesn't just mean print-on-the-screen. It also means 'take a new line', and start at the left-hand side! You will also find, incidentally, that when the words on the screen reach the bottom line of the display section, then all the lines appear to move up, and the top line disappears. This is the action that is called 'scrolling', and it's the way that the machine deals with displaying lots of lines on a screen which holds only 24 lines altogether, not counting the bottom lines which are used for looking at commands that you are typing. Later, in Chapter 8, we'll look at a command which causes scrolling.

Now the action of selecting a new line for each PRINT isn't always convenient, and we can change the action by using punctuation marks that we call print modifiers. Start this time by acquiring a new habit. Type NEW and then press the ENTER key. This clears the old program out, and you might also like to use the CLS action to clear the screen. If you don't use the NEW action, there's a chance that you will find lines of old programs getting in the way of new ones. Each time you type a line, you delete any line that had the same line number in an older program, but if there is a line number that you don't use in the new program it will remain stored. In Fig. 2.2, for example, the line 15 that you added would be left in store even when you typed a new line 10 and a new line 20. If you are using a printer with your QL, you will have to

remember that the PRINT#3 instruction (or whatever you have used) will no longer operate after a NEW, and you will have to use OPEN #3,SER1 (or similar) to re-establish printing.

Now try the program in Fig. 2.4. There's a very important difference between Fig. 2.4 and Fig. 2.3, as you'll see when you RUN it. The effect of a semicolon following the last quote in a line is to prevent the next

```
10 PRINT"This is ";
20 PRINT"the remarkable ";
30 PRINT"QL Computer."
```

Fig. 2.4. The effect of semicolons.

piece of printing starting on a new line at the left-hand side. When you RUN this program, all of the words appear in one line. It would have been a lot easier just to have one line of program that read:

```
10 PRINT "This is the remarkable QL Computer."
```

to do this, but there are times when you have to use the semicolon to force two different print items on to the same line. We'll look at that sort of thing in later program examples.

Rows and columns

Neat printing is a matter of arranging your words and numbers into rows and columns, so we'll take a closer look at this particular art now. To start with, we know already that the instruction PRINT will cause a new line to be selected, so the action of Fig. 2.5 should not come as too much of a surprise. Lines 10 and 20 contain a novelty, though, in the

```
10 CLS:PRINT"This is the QL"
20 PRINT:PRINT
30 PRINT"Ready to work for you."
```

Fig. 2.5. Clearing the screen with the CLS instruction, and using multistatement lines. PRINT is being used to make a blank line.

form of two instructions in one line. The instructions are separated by a colon (:), and you can, if you like, have several instructions following one line number in this way, taking several screen lines. The only practical limit to this is that it makes your instructions too hard to read if you put too many instructions together in this way. In a 'multistatement' line of this type, the QL will deal with the different

instructions in a left-to-right order. The instruction CLS should not surprise you either – this clears the screen, and makes the printing start at the top left-hand corner. It's the same action as the CLS direct command, but done automatically within the program.

Another point about Fig. 2.5 is that line 20 causes the lines to be spaced apart. The two PRINT instructions, with nothing to be printed, each cause a blank line to be taken. There are other ways of doing this, as we'll see, but as a simple way of creating a space, it's very handy.

Figure 2.6 deals with columns. Line 10 is a PRINT instruction that acts on the numbers 1 and 2. When these appear on the screen,

```
10 PRINT 1,2
20 PRINT 1,2,3,4
30 PRINT"ONE","TWO"
40 PRINT"ONE","TWO","THREE","FOUR"
50 PRINT"THIS ITEM IS LONGER","TWO"
60 PRINT 1,2,3,4,5,6
```

Fig. 2.6. How the comma causes words to be placed into four columns.

though, they appear spaced out just as if the screen had been divided into columns. The mark which causes this effect is the comma, and the action is completely automatic. The comma is on the key next to the letter 'M', and if you use the apostrophe on the key next to ENTER, you will not get the same effect! The two look rather alike on the keyboard, but completely different on the screen. On my QL, the use of the apostrophe caused an error message, though the Provisional Manual stated that it would cause a new line to be taken. As line 20 shows, you can get four columns, each one of which allows room for eight characters. Anything that you try to get into a fifth column will actually appear on the first column of the next line down. The action works for words as well as for numbers, as lines 30 and 40 illustrate. When words are being printed in this way, though, you have to remember that the commas must be placed *outside* the quotes. Any commas that are placed inside the quotes will be printed just as they are and won't cause any spacing effect. You will also find that if you attempt to put into a column something that is too large to fit, the long phrase will spill over to the next column, and the next item to be printed will be at the start of the next line. Line 50 illustrates this – the first phrase spills over from column 1 all the way to column 3, and the word TWO is printed starting at column 4 on the same line. Line 60 shows what happens if you keep using commas – the columns just take up the same positions on the next line.

Commas are useful when we want a simple way of creating four columns. A much more flexible method of placing words on the screen exists, however. This is programmed by using the command word AT, and it allows you to place words on the screen at any position, and also in any order. Normally, when you PRINT on to the screen, a PRINT instruction causes the computer to take a new line and you cannot go back to the previous one. By using AT, you can place words and numbers where you like, and even have one line replace another if you want to.

Now if you have programmed a Spectrum, you may think that you know all about the use of AT. However, the QL uses AT in rather a different way, so don't skip this section! The important difference is that the QL AT command *must* be issued before the PRINT instruction that it refers to. It can be in the line just before the PRINT instruction, or it can be in the same line, separated by a colon. You *cannot*, however, place the AT *after* the PRINT as you do with Spectrum, or any of the other computers that use this type of command.

AT has to be followed by two numbers. Of these, the first number is a 'column' number. You can print 37 characters (letters, numbers, punctuation marks) in a line across the screen of the QL. Each of these characters is in one column, because characters in all of the other lines are also spaced out in the same way. We can use a number, then, to represent the position of a character on a line. The number can range from 0 (left-hand side) to 36 (right-hand side) – a total of 37 columns. The second number of the AT command is a 'line' number. The lines on the screen are numbered starting with 0 (the top line), and ending with 19 (bottom line of the display part of the screen) – a total of 20 lines.

An example would certainly help here. Take a look at Fig. 2.7. Line 10 clears the screen, and line 20 introduces the first AT. This is AT 0,0

```

10 CLS
20 AT 0,0:PRINT"TOP LEFT"
30 PAUSE 100
40 AT 25,19:PRINT"BOTTOM RIGHT"
50 PAUSE 100
60 AT 0,19:PRINT"BOTTOM LEFT"
70 PAUSE 100
80 AT 28,0:PRINT"TOP RIGHT"
90 PAUSE 100
100 AT 16,9:PRINT"MIDDLE"

```

Fig. 2.7. How AT is used to position the cursor so that you can print anywhere on the screen.

meaning that printing will start on the top left-hand corner of the screen. There *must* be a space between the 'T' of AT and the first number. The 'T' of TOP LEFT will therefore be placed at the top left corner of the screen, as it would anyhow just following a CLS. The next line then causes a pause. The word PAUSE, followed by a number, causes a time delay. The number is the delay time in units of $\frac{1}{50}$ seconds ($\frac{1}{60}$ in the US version), so that the number 50 would give a delay of one second, 100 gives two seconds and so on.

The next AT is followed by 25,19, so that the phrase 'BOTTOM RIGHT' is placed with the 'T' of RIGHT in the bottom right-hand corner of the screen. How was this calculated? The answer is by counting the 'T' of right as column 36, then counting back 35,34,33 and so on until I reached the 'B', which was at 25. This is to be on the bottom line, number 19. With this much information now, you can see how the rest of the words have been put into position. Because of the pauses, you can see that we are not following the normal order of left-to-right, top-to-bottom for printing.

Oh, yes, how did I position the word 'MIDDLE' at the centre of a line in Fig. 2.7? This is done by calculating the correct numbers for the AT instruction. The method is shown in Fig. 2.8. You have to count up the number of characters that you want to print centred. By

-
1. Count number of characters in the title, including spaces.
 2. Subtract this number from 38.
 3. Divide the result by 2, ignoring any remainder.
 4. Use the result as the first part of the AT number. (For perfect centring on your TV receiver, you may have to add or subtract 1 from this number.)
-

Fig. 2.8. The formula for centring a title.

'characters', I mean letters, digits, spaces and punctuation marks. You then subtract this from 38, and divide the result by 2. Take the whole number part of the answer – forget about any half left over – and this is then the correct column number to use with AT. In this example, we also wanted to place the word in the central line. With an even number of lines, there is no central line, so we can use either line 9 (too high) or line 10 (too low). You will see, in a later chapter, that we can use letters in place of numbers in the AT (and other) instructions. This allows us to centre words without all the fuss of counting letters – but that's more advanced programming than we should be thinking about at this point!

Figure 2.9 is a map that you will find useful for placing words where

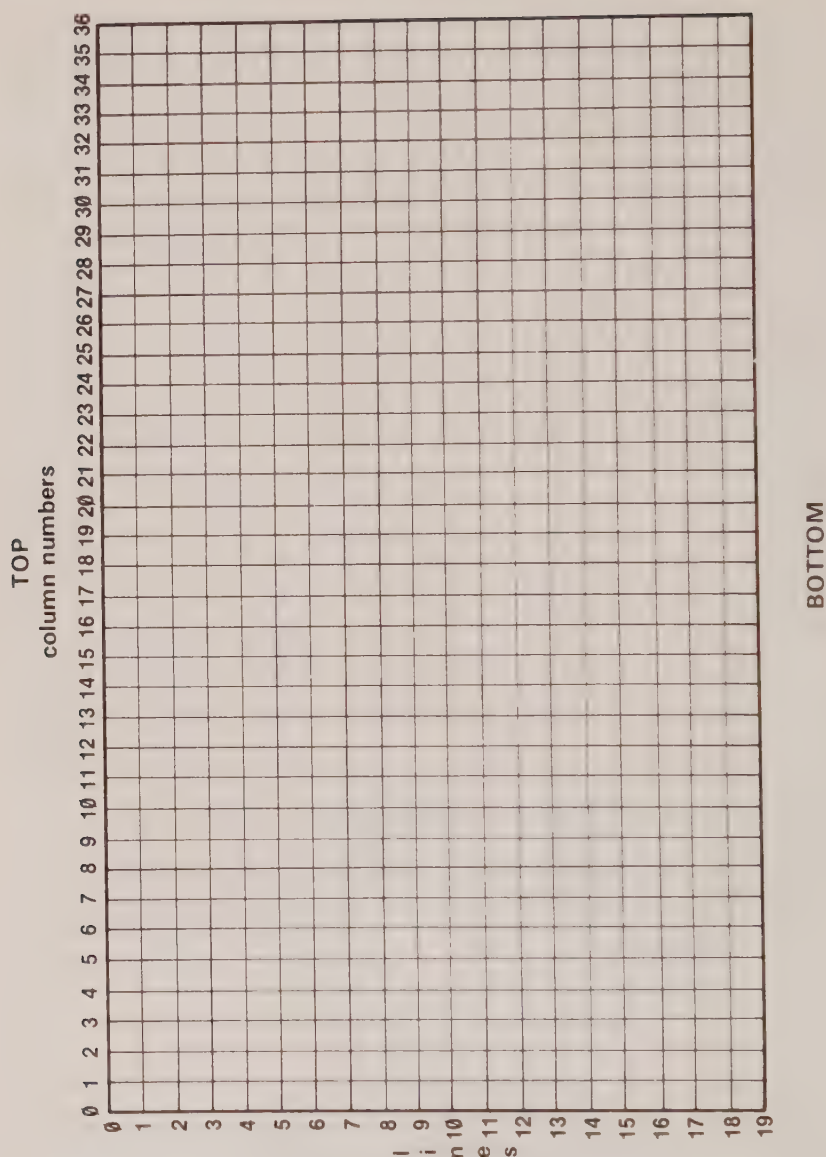


Fig. 2.9. The AT map, showing how the AT numbers correspond to points on the screen.

you want them on the screen. Meanwhile, there's more to learn about printing characters. You will find that when you first start programming for yourself, the appearance of words on the screen leaves a lot to be desired. You will, in particular, find that words are split from one line to the next in a very untidy way. The QL offers you

help with this in the shape of the 'forward slash' character, which is on the key on the extreme right-hand side, top row. If you place this character following a PRINT, it will cause a new line to be taken. This allows you to organise all of your printing in one command. If you have a line that starts normally with PRINT", then you type words until you can see that the next word that you type will go to a column beyond the one which contains the quotemarks. At this point, you can put in quotemarks, then the forward slash sign, then more quotes – and continue! Provided you never type a word across the column which contains the quotemark for the previous line, your printing will look neat.

I'll give an example of that in a moment, but there's another printing trick of the QL to look at first. It's a good one – control of character size. This is done by using the CSIZE command. CSIZE has to be followed by two numbers. The first of these is a width number, the second is a height number, and you are restricted to a few numbers only. As an example of this in action, look at Fig. 2.10. This clears the

```
10 CLS
20 CSIZE 3,1
30 AT 9,2:PRINT"BIG TITLE"
40 CSIZE 2,0
50 AT 2,8:PRINT"This shows how you ca
n control the\"size of characters."
```

Fig. 2.10. Using CSIZE to change the character size on the screen.

screen in line 10, and then chooses a new character size in line 20. This is the maximum size that we can obtain, and it's useful for titles. In this size of character, the width is one third greater than normal for a TV display ($\frac{8}{3}$ times greater than a monitor letter width), and the height is twice normal. Because of the different width, we have had to use smaller numbers in the AT command in line 30. We work on 28 characters per line, and only 10 lines per screen height. The words, 'BIG TITLE' are therefore printed in these large characters by line 30. Line 40 restores normal print, and the AT in line 50 has to be chosen so that the words do not cover the title. You will see the forward-slash sign being used here to get the words fitting neatly into the lines.

The height number for CSIZE can be 0 or 1 only, but you can choose a range of 0 to 3 for the width. Figure 2.11 indicates how these different choices look on the screen. The manual shows a choice of four widths, but you will see only two widths and two heights *for the characters on*

```
10 CLS
20 CSIZE 0,0:PRINT"SIZE 0,0"
30 CSIZE 1,0:PRINT"SIZE 1,0"
40 CSIZE 2,0:PRINT"SIZE 2,0"
50 CSIZE 3,0:PRINT"SIZE 3,0"
60 CSIZE 0,1:PRINT"SIZE 0,1"
70 CSIZE 1,1:PRINT"SIZE 1,1"
80 CSIZE 2,1:PRINT"SIZE 2,1"
90 CSIZE 3,1:PRINT"SIZE 3,1"
```

Fig. 2.11. Looking at the range of character sizes. You can see the complete range only if you have selected the F1 option.

the TV screen. If you use a monitor, you will find the full range of sizes that the manual describes. If you want to see the whole range on a TV screen, type MODE 512 (then ENTER) and run the program again. After you have looked at the results, type MODE 256 (ENTER) again to return to a normal TV display. For most of us, using a TV receiver (all of us, until the monitor cables become available!), the choice is of two widths, using 0 or 2 for the small width and 1 or 3 for the large one. For height, 0 selects small height, and 1 selects large height.

Chapter Three

A Feast of Variables

So far, our computing has been confined to printing numbers and words on the screen. That's one of the main aims of computing, but we have to look now at some of the actions that go on before anything is printed. One of these is called *assignment*. Take a look at the program in Fig. 3.1. Type it in, run it, and contrast what you see on the screen

```
10 CLS
20 LET X=23
30 PRINT"2 TIMES ";X;" IS ";2*X
40 X=5
50 PRINT"X IS NOW ";X
60 PRINT"AND TWICE ";X;" IS ";2*X
```

Fig. 3.1. Assignment in action. The letter X has been used in place of a number.

with what appears in the program. The first line that is printed is line 30. What appears on the screen is:

```
2 TIMES 23 IS 46
```

but the numbers 23 and 46 don't appear in line 30! This is because of the way we have used the letter X as a kind of code for the number 23. The official name for this type of code is a 'variable name'.

Line 20 assigns the variable name X, giving it the value of 23. Assigns means that wherever we use X, *not* enclosed by quotes, the computer will operate with the number 23 instead. Since X is a single character and 23 has two digits, that's a saving of space. It would have been an even greater saving if we had assigned X differently, perhaps as X=2174.3256, for example. Line 30 then proves that X is taken to be 23, because wherever X appears, not between quotes, 23 is printed, and the 'expression' 2*X is printed as 46. We're not stuck with X as

representing 23 for ever, though. Line 40 assigns X as being 5, and lines 50 and 60 prove that this change has been made.

That's why we call X a 'variable' -- we can vary whatever it is we want it to represent. Until we do change it, though, X stays assigned. Even after you have run the program of Fig. 3.1, providing you haven't added new lines or deleted any part of it, you can type PRINT X, and pressing ENTER will show the value of X on the screen. The listing also shows that this action of assignment can be carried out in two ways. One of these ways (line 20) uses the instruction word LET, the other (line 40) does not. The QL permits you to use either method. Since LET requires you to type three more letters and a space, it's just as well to use the shorter and more direct form.

This very useful way to handle numbers in code form can use a 'name' which must start with a letter, upper-case (capital) or lower-case (small). You can add to that letter other letters, making a complete word if you like, or digits, but not spaces, arithmetic symbols (+, -, *, /) or punctuation marks. You *can* use the underline mark as much as you like, however. Names like TOTAL, lastname, ALL_THERE_IS and R2D2 can all be used for number variables, and each can be assigned to a different number. One thing that you need to be careful about, though, is that *the QL does not distinguish between upper and lower case names*. If you assign the variable name of 'jam' to the number 45, and then assign the name of 'JAM' to 88, you will find that both 'jam' and 'JAM' have been assigned to the same value. This will be whichever number you assigned most recently. Another thing to watch for is the use of 'reserved words'. The reserved words of the QL are its instruction words, words like PRINT, NEW, RUN and so on. You *cannot* use these as variable names, and you will get a 'bad line' error message if you attempt to use them. Some computers won't even allow you to use words which *contain* these reserved words, so that you could not use words like 'NEWLY'. The QL, however, is more tolerant, and will allow you to use any word which is not identical to a reserved word.

Just to make it even more useful, you can use similar 'names' to represent words and phrases also. The difference is that you have to add a dollar sign (\$) to the variable name. If N is a variable name for a number, then N\$ (pronounced en-string or en-dollar) is a variable name for a word or phrase. The computer treats these two, N and N\$, as being entirely separate and different. They also have to be assigned in rather different ways. When you assign a number to a number variable, using LET or just the = sign, you don't have to type a quotemark (") on each side of the number. When you assign a string variable in this way, however, you have to make use of quotemarks. We'll look at other methods of making assignments later.

Serenade for strings

Figure 3.2 illustrates ‘string variables’, meaning the use of variable names for words and phrases. Lines 10 and 20 carry out the assignment operations, and lines 30 to 50 show how these variable names can be used. Notice that you can mix a variable name, which doesn’t need quotes around it, with ordinary text, which *must* be surrounded by quotes. You have to be careful when you mix these two, because it’s easy to run words together. Note in lines 30 to 50 how spaces have been left between words. When you are printing one variable after another,

```
10 CLS:NAME$="QL"
20 FIRST$="The remarkable":LAST$="Com
  puter system"
30 PRINT FIRST$;" ";NAME$;" ";LAST$
40 PRINT "This uses the ";NAME$
50 PRINT FIRST$;" ";NAME$;" in action
  !"
```

Fig. 3.2. Using string variables. These are distinguished by the dollar sign.

the space is created by typing quotes, then pressing the spacebar, then another quotemark, like “ ”. To leave a space between text in quotes and a variable name, you only need to press the spacebar at the point where you need the space. The semicolons are *essential* when you are joining up bits of text in this way. If you omit the semicolon at a join, you will get a ‘bad line’ message.

Figure 3.3 shows another example, this time using the variable names BLURB\$ and puff\$ for longer phrases. For some quaint reason,

```
10 CLS:BLURB$="The new computer"
20 puff$="that brings more power at l
  ess cost"
30 PRINT "QL _ "
40 PRINT BLURB$!puff$
50 PRINT BLURB$;" ";puff$
```

Fig. 3.3. Illustrating variable names with more than two letters. These can be very useful to remind you of what the name means.

even though I typed PUFF\$, the computer insisted on giving me ‘puff\$’, though the upper-case letters in BLURB\$ were accepted. This might have been one of the minor eccentricities of an early machine, and it was a curious exception to the general rule that the machine

converted names from lower-case to upper-case. It won't convert anything that you have typed between quotes, though. There wouldn't be much point in printing messages in this way if you wanted the message once only, but when you continually use a phrase in a program, this is one method of programming it so that you don't have to keep typing it! Oh, yes, there's another odd thing here. In line 40, the exclamation mark has been used to separate the variable names. This is a unique feature of the QL, an 'intelligent spacer'. When you use the exclamation mark as a spacer between variables in this way, the printing on the screen will try to avoid splitting words. If the 'puff\$' phrase were printed directly following BLURBS on the same line, the word 'power' would be cut in half, as line 50 shows. If, of course, you make one of the variables equal to a line of text that takes more than one line on the screen, the use of the exclamation mark won't prevent a word from being split.

Strings and things

Because the name of a string variable is marked by the use of the \$ sign, a variable like A\$ is not confused with a number variable like A. We can, in fact, use both on the same program knowing that the computer at least will not be confused. Figure 3.4 illustrates that the difference is a bit more than skin deep, though. Lines 10 and 20 assign number

```

10 A=2:B=3
20 A$="2":B$="3"
30 CLS
40 PRINT A;" TIMES ";B;" IS ";A*B
50 PRINT"NUMBERS ARE ";A$;" AND ";B$
60 PRINT A$;" TIMES ";B$;" IS ";A$*B$
70 REMARK IMPOSSIBLE ON OTHER COMPUTE
RS- YOU WOULD GET AN ERROR MESSAGE!
80 C$="BLUE":D$="VIOLET"
90 PRINT C$*D$
100 REMARK THIS DOESN'T WORK!
```

Fig. 3.4. String and number variables might look alike when they are printed, but they are different!

variables A and B, and string variables A\$ and B\$. When these variables are printed, you can't tell the difference between A and A\$ or between B and B\$. The QL, unlike other computers, is arranged to deal

with numbers in the same way, no matter whether they are in the form of number variables or string variables, as line 60 shows. The difference appears, however, when the computer attempts to complete line 90. It can multiply two number variables, or two strings that represent number variables, because numbers can be multiplied, but it can't multiply string variables that aren't numbers. You can multiply "2" by "3", but you can't multiply "2 LABURNUM WAY" by "3 ACACIA AVENUE". The computer therefore refuses to carry out multiplication, division, addition, subtraction or any other arithmetic operation on strings that contain anything which isn't a number. Attempting to do a forbidden operation in line 90 causes an error message when the program runs, and this error will always halt a program. The message that appears is 'At line 90 error in expression' and it means that you have tried to do something that the QL does not permit. Later on, we'll see that there are operations that we can carry out on strings that we can't carry out on numbers, and attempts to do these operations on numbers will also cause the same error message. The difference is an important one. The computer stores numbers in a way that is quite different from the way it stores strings. The different methods are intended to make the use of arithmetic simple for number variables (for the computer, that is), and to make other operations simple for strings. Let's face it, it's only a machine!

On the QL, the '+' sign will *always* cause addition if addition is possible, and you will get an error message if you try to add strings that do not consist of numbers. There is one operation of joining that can be carried out on strings, but not on numbers. It uses the & sign, but it isn't addition in the sense of adding numbers. Figure 3.5 illustrates this action of joining strings, which is often called *concatenation*. This is

```

10 A$="PONSONBY"
20 B$="BROWN"
30 CLS
40 PRINT"Just call me ";A$&"-"&B$;" ,
   he said."
50 PRINT:A$="123":B$="456"
60 PRINT"Joined string is ";A$&B$
70 PRINT"Addition sign gives ";A$+B$

```

Fig. 3.5. Concatenating or joining strings. This is not the same action as addition!

nothing like the action of arithmetic, as you'll see by lines 40 and 60. Line 60 uses numbers in place of the names placed between the quotes.

Just to point out the difference, line 70 shows what happens if a '+' sign is used on the numbers in string form. Concatenation is a very useful way of obtaining strings which otherwise would need rather a lot of typing. Take a look at Fig. 3.6. This defines strings A\$ and B\$ as characters which can be used as 'frames' around a title. The title is defined in line 20 as 'THE NEW QL'. Line 40 then prints a concatenated string.

```
10 A$="***":B$="###"
20 S$="THE NEW QL"
30 CLS
40 PRINT A$&B$&S$&B$&A$
```

Fig. 3.6. Using concatenation to make a frame for a title.

Along with this business of concatenation, there's a very useful command which will join a number of identical characters into a string for you. The command is FILL\$, and it has to be followed by two items, enclosed in brackets. The first item is the character that you want to use, and the second item is the number of these characters you want. For example, if you program G\$=FILL\$("\$",20), this will make the string G\$ contain twenty dollar signs. Figure 3.7 illustrates this FILL\$ action used to make a frame for a title.

```
10 CLS
20 A$=FILL$("*",10)
30 B$=FILL$("#",5)
40 AT 1,2:PRINT A$&B$&"TITLE"&B$&A$
```

Fig. 3.7. Using FILL\$ to make a string of identical characters.

Getting some input

So far, everything that has been printed on the screen by a program has had to be placed in the program before it is run. We don't have to be stuck with restrictions like this, however, because the computer allows us another way of putting information, number or name, into a program while it is running. A step of this type is called an INPUT and the BASIC instruction word that is used to cause this to happen is also INPUT.

Figure 3.8 illustrates this with a program that prints your name. Now

I don't know your name, so I can't put it into the program beforehand. What happens when you run this is that the words:

What is your name

are printed on the screen. On the line below this you will see the flashing (blue) cursor. The computer is now waiting for you to type something,

```
10 CLS
20 PRINT "What is your name"
30 INPUT NAME$
40 CLS:PRINT:PRINT
50 PRINT NAME$;" -this is your life!"
```

Fig. 3.8. Using the INPUT instruction. The name that you type is put into the phrase in line 50.

and then press ENTER. Until the ENTER key is pressed, the program will hang up at line 30, waiting for you. If you're honest, you will type your own name and then press ENTER. You don't have to put quotes around your name; simply type it in the form that you want to see printed. When you press ENTER, your name is assigned to the variable NAME\$. The program can then continue, so that line 40 clears the screen and spaces down by two lines. Line 50 then prints the famous phrase with your name at the start. You could, of course, have answered Mickey Mouse or Donald Duck or anything else that you pleased. The computer has no way of knowing that either of these is not your true name. Even if you type nothing, and just press ENTER, it will carry on, with no name at all. Don't listen to the nutters who tell you that computers know everything!

We aren't confined to using string variables along with INPUT. Figure 3.9 illustrates an INPUT step which uses a number variable *n*.

```
10 CLS:PRINT "Enter a number, please"
20 INPUT n
30 CLS:PRINT
40 PRINT "Twice ";n;" is ";2*n
```

Fig. 3.9. An INPUT to a number variable. The quantity that you type must be a number.

The same procedure is used. When the program hangs up with the cursor appearing, you can type a number and then press the ENTER key. The action of pressing ENTER will assign your number to *n*, and allow the program to continue. Line 40 then proves that the program is

dealing with the number that you entered. When you use a number variable in an INPUT step, then what you have typed when you press ENTER must be a number. If you attempt to enter a string, the computer will refuse to accept it, and you will get an error message - 'At line 20 error in expression'. If your INPUT step uses a string variable, then *anything* that you type will be accepted when you press ENTER, but you will get an error message if you try to perform arithmetic on something that is not a number.

The way in which INPUT can be placed in programs can be used to make it look as if the computer is paying some attention to what you type. Figure 3.10 shows an example - but with INPUT used in a

```

10 CLS
20 INPUT"Type your name, please ";NM$
30 PRINT
40 PRINT"Very pleased to meet you, ";
NM$

```

Fig. 3.10. Using INPUT to print a phrase which requests the input.

different way. This time, there is a phrase following the INPUT instruction. The phrase is placed between quotes, and is followed by a semicolon and then the variable name NM\$. This line 20 has the same effect as the two lines:

```

15 PRINT "Type your name, please";
20 INPUT NM$

```

and this time the cursor appears on the same line as the question, and your reply is also on the same line - unless the length of the name causes letters to spill over on to the next line. You can also use the comma or exclamation marks in place of the semicolon in a line like this. Try the effect for yourself. The comma causes the name that you enter to be placed at the start of the next screen column, but the apostrophe did not have any noticeable effect on my QL. As you know already, the forward slash sign can be used to select a new line, and it has the same effect when used in place of the semicolon in this example. Another variation on INPUT is the use of 'channels'. You will have noticed how the screen is divided into sections. If you use code numbers, called channel numbers, following a PRINT or INPUT instruction, you can specify which section of the screen you use. Try changing the INPUT in Fig. 3.10 into INPUT#0, (and don't forget the comma!). You will find that this causes the message and your answer to appear at the bottom

section of the screen. Using INPUT#2, is the same as plain INPUT, but INPUT#3, causes the input to appear where a program would be listed. On my QL, these uses of INPUT with channels caused a flashing line to appear on the screen also. Some other variations also caused the machine to seize up, so always keep a recording of anything new.

The use of INPUT isn't confined to a single name or number. We can use INPUT with two or more variables, and we can mix variable types in one INPUT line. Figure 3.11, for example, shows two variables

```
10 CLS
20 INPUT"Name and number, please";NM$
,N
30 PRINT:PRINT
40 PRINT"The name is ";NM$
50 PRINT"The number is ";N
```

Fig. 3.11. Putting in two variables in one INPUT step.

being used after one INPUT. One of the variables is a string variable NM\$, the other is the number variable N. Now when the computer comes to line 20, it will print the message and then wait for you to enter both of these quantities, a name and then a number. There is just one way of doing this correctly – that is, to type the name, and then press ENTER. The computer will then shift the cursor to the next line. This means 'more needed', and that's a signal for you to type the number and then press ENTER again. The name and number will then be printed again in lines 40 and 50. If you use an exclamation mark between NM\$ and N, instead of a comma, then both the name and the number can be entered in the same line.

We can extend this principle further. Figure 3.12 calls for four numbers to be entered. These must be entered one by one, pressing

```
10 CLS
20 INPUT"Four numbers, please ";A,B,C
,D
30 PRINT
40 PRINT"The sum of these is ";A+B+C+
D
```

Fig. 3.12. An INPUT step which calls for four numbers. These must be entered one by one.

ENTER each time. Once again, the numbers are assigned to the variable names, and the program will print the sum in line 40. Note,

incidentally, that when you assign a name to a string variable, as when you type your name to be assigned to NM\$, you *don't* have to use quotemarks.

Reading the data

There's yet another way of getting data into a program while it is running. This one involves reading items from a list, and it uses two instruction words READ and DATA. The word READ causes the program to select an item from the list. The list is marked by starting each line of the list with the word DATA. The items of the list can be separated by commas. Each time an item is read from such a list, a 'pointer' is altered so that the next time an item is needed, it will be the next item on the list rather than the one that was read the last time round.

We'll look at this in more detail in Chapter 5, but for the moment we can introduce ourselves to the READ...DATA instructions. Figure 3.13 uses the instructions in a very simple way. Line 20 reads an item number, which is the first item on the list and assigns it to the variable J.

```

10 CLS
20 READ J
30 PRINT" Item ";J;
40 PRINT" is a ";
50 READ N$
60 PRINT N$
80 DATA 5,"disk drive"
```

Fig. 3.13. Using the READ and DATA words to place information into a program.

This is printed in line 30, with the semicolon keeping printing in the same line so that the phrase in line 40 follows it. The semicolon at the end of line 40 once more keeps the printing in the same line, and line 50 reads the name which is the second item in the list. This is assigned to the variable name N\$ and printed in line 60. Line 80 contains the DATA, one number and one phrase. The number can be typed as it is, but you can see that the phrase needs quotes round it. Anything that you assign to a string variable in this way must have quotes around it in its DATA line. You always have to be careful about how you match your READ and your DATA. If you use a number variable in the READ, like READ A, then what is in the DATA line being read *must*

be a number. If you use a string variable, as in `READ A$`, then it doesn't matter whether your `DATA` line contains a number or a string. Remember that if you read a number using `READ A$`, then the QL allows you to carry out any arithmetic on that number.

Now for an odd experience. If you have typed the program of Fig. 3.13, and run it, you will have seen it operate. Now type `RUN` again, and press `ENTER`. This time, instead of the program running, you get an error message that says:

At line 20 end of file

What does this mean? Quite simply, it means that you have read both of the items in the data line, and there aren't any more! Unlike other computers, the QL does not start its `DATA` list all over again each time you use `RUN`. There are two ways round this. One is to type `CLEAR` (and press `ENTER`) before a `RUN`, the other is to have a single line instruction:

5 RESTORE

added to the program. `RESTORE` means 'go to the start of the `DATA` list again', and it's an instruction that we'll be looking at again, because it has other uses. Meanwhile, though, you should remember that any program which makes use of `READ...DATA` should start with a `RESTORE`.

The `READ...DATA` instructions really come into their own when you have a long list of items that are read by repeating a `READ` step. These would be items that you would need every time that the program was used, rather than the items you would type in as replies. We're not quite ready for that yet, so having introduced the idea, we'll leave it for now.

Number antics

The amount of computing that we have done so far should have persuaded you that computers aren't just about numbers. For some applications, though, the ability to handle numbers is very important. If you want to use your computer to solve scientific or engineering problems, for example, then its ability to handle numbers will be very much more important than if you bought it for games, for word processing or even for accounts. It's time, then, to take a very brief look at the number abilities of the QL. It is a brief look because we simply don't have the space to explain what all the mathematical operations

do. In general, if you understand what a mathematical term like *sin* or *tan* or *exp* means, then you will have no problems about using these mathematical functions in your programs. If you don't know what these terms mean, then you can simply ignore the parts of this section that mention them.

The simplest and most fundamental number action is counting. Counting involves the ideas of incrementing if you are counting up and decrementing if you are counting down. Incrementing a number means adding 1 to it, decrementing means subtracting 1 from it. These actions are programmed in a rather confusing-looking way in BASIC, as Fig. 3.14 shows. Line 20 sets the value of variable X as 5. This is printed in line 30, but then line 40 'increments X'. This is done using the odd-looking instruction : X = X + 1, meaning that the new value that is assigned to X is 1 more than its previous value. The rest of the program proves that this action of incrementing the value of X has been carried out.

```

10 CLS
20 X=5
30 PRINT"Value of X is ";X
40 X=X+1:PRINT
50 PRINT"Now we've used X=X+1":PRINT
60 PRINT"The value of X is ";X

```

Fig. 3.14. Incrementing, using the equals sign to mean 'becomes'.

The use of the = sign to mean 'becomes' is something that you have to get accustomed to. When the same variable name is used on each side of the equality sign, this is the use that we are making of it. We could equally well have a line:

$$X = X - 1$$

and this would have the effect of making the new value of X one less than the old value. X has been *decremented* this time. We could also use $X = 2 * X$ to produce a new value of X equal to double the old value, or $X = X / 3$ to produce a new value of X equal to the old value divided by three. Figure 3.15 shows another assignment of this type, in which both a multiplication and an addition are used to change the value of X.

Figure 3.16 illustrates the use of some number functions. A number function in this sense is an instruction which operates on a number to produce another number. Line 10 picks the value of 2.5 for X. Line 20 then prints the value of X squared, meaning X multiplied by X. This is

```

10 CLS
20 X=5:PRINT"X IS ";X
30 PRINT
40 X=2*X+4
50 PRINT"It's changed - "
60 PRINT"X is now ";X

```

Fig. 3.15. A more elaborate reassignment, using an 'expression'.

```

10 CLS:X=2.5
20 PRINT"X squared is ";X^2
30 PRINT
40 PRINT"Its square root is ";SQRT(X)
50 PRINT
60 PRINT"Its natural log is ";LN(X)
70 PRINT"and the ordinary log is ";LOG
10(X)

```

Fig. 3.16. Using some number functions.

programmed by typing X^2 , and the character which the QL uses for this is on the '6' key. To get the square root of the number that has been assigned to X, we use the instruction word SQRT. An alternative is $X^{.5}$, but SQRT(X) is easier to type and remember. For other roots, you can use expressions like $X^{(1/3)}$ for the cube root and so on. LN(X) produces the natural logarithm of X. This is not the type of logarithm that you may want, and to find the ordinary (base 10) log, you have to use LOG10(X).

Figure 3.17 illustrates the various number functions that can be used, with a brief explanation of what each one does. Some of these actions will be of use only if you are interested in programming for scientific, technical or statistical purposes. Others, however, are useful in unexpected places, such as in graphics programs.

How precise?

One of the problems of small computers is precision of numbers. You probably know that the fraction $\frac{1}{3}$ cannot be expressed exactly as a decimal. How near we can get to its true value depends on the number of decimal places we are prepared to print, so that 0.33 is closer than 0.3, and 0.333 is closer still. The computer converts most of the numbers it works with into the form of a fraction and a multiplier. The

ABS (X) Converts negative sign to positive.
 ACOT(X) Gives the angle (in radians) whose cotangent is X.
 ATN (X) Gives angle (in radians) whose tangent is X.
 COS (X) Gives the cosine of angle X (radians).
 COT(X) Gives the cotangent of angle X (radians).
 DEG(X) Converts angle X, in radians, to degrees.
 DIV Performs an integer division, gives integer result.
 EXP(X) Gives the value of e to the power X.
 INT(X) Gives the whole-number part of X.
 LN(X) Gives the natural logarithm of X.
 LOG10(X) Gives the common logarithm of X.
 MOD Gives the integer remainder after integer division.
 PI Gives a value for the number pi.
 RAD(X) Converts angle X (degrees) into radians.
 RANDOMISE Sets random number generator to another set of numbers.
 RND Gives a randomly calculated fraction.
 RND(X TO Y) Calculates integers at random, lying between, and including, X and Y.
 SIN(X) Gives the sine of angle X (radians).
 SQRT(X) Gives the square root of X.
 TAN(X) Gives the value of the tangent of angle X (radians).

Fig. 3.17. QL number functions, with brief notes. Don't worry if you don't know what some of these do. If you don't know, you probably don't need them!

fraction is not a decimal fraction but a special form called a binary fraction, and this conversion is seldom exact. The conversion is particularly awkward for numbers like 1, 10, 100 and also .1, .01, .001; all the powers of ten, in fact. To avoid embarrassments like printing $3 - 2 = .9999999$, the computer will round numbers of this type up or down as need be before displaying them. Not all computers do this well - you can be glad that you bought a QL! Figure 3.18 shows how

```

10 CLS
20 PRINT 1/3, 2/3
30 PRINT 1/11, 10/11
40 PRINT 1/3+2/3, 1/11+10/11

```

Fig. 3.18. How the QL deals with awkward fractions.

the QL copes with fractions like $\frac{1}{3}$, $\frac{2}{3}$, $\frac{1}{11}$ and $\frac{10}{11}$. The numbers that you see on the screen have been rounded to seven places of decimals, but the number that the computer *stores* must be of many more

decimal places. The rounding works to make sure that adding the fractions gives the correct result.

You can also see from how the computer prints the fraction $\frac{1}{11}$ that there is an alternative method of printing numbers. If you have worked in any subject that uses very large or very small numbers (Physics, Chemistry, Engineering), you will know about this method already. If you haven't met it before, it's called *standard form*. A number in standard form consists of a quantity which lies somewhere between 0 and 10, never quite equal to either of these, and multiplied by a power of ten. Take, for example, a number like 132000. If we shift the decimal point five places *left* (equivalent to *dividing* by ten to the power 5), then this becomes 1.32. To get the value correct, this would have to be multiplied by 10 to the power 5, or, as we write it E5. The number 132000 could therefore be written as 1.32E5.

Suppose we try a small number, 0.00036. This is 3.6 times ten to the minus 4, or 3.6E-4, with the minus sign there because we have had to shift the decimal point four places *right* to get this result. The QL, like most small computers, will accept and print numbers in this form. The conversion is automatic, and the QL will display numbers in this form only when it has to – which means when the numbers would need more than seven figures to display.

Most computers allow a limited range of numbers to be stored in a much more precise way, called an *integer variable*. An integer, as far as the QL is concerned, is a whole number whose value lies between the limits of -32768 and +32767. An integer variable name consists of the variable name followed by the % sign. If you assign a number to an integer variable, then only numbers in the correct range can be used, and any fractions will be discarded. You will get the error message 'Overflow' if you try to do something like:

```
A%=32800
```

for example. Figure 3.19 illustrates the action of rejecting fractions, because when the variable X, whose value is 3.7, is assigned to X% in line 30, then printing X% in line 40 gives 4 only. The fraction .7 has been rounded up to 1. This is unusual, and most other computers, given this line, would simply omit the .7, giving 3.

The advantage of using integer variables is twofold. One advantage is that any arithmetic, apart from division, that we carry out on integers is exact, with no rounding up or down needed. Division is the exception because fractions are ignored, as line 70 of Fig. 3.19 shows. The other advantage of integers is that they need less memory to store. A program that uses integers will also run much faster than one which uses any


```

10 CLS:X=3.7
20 PRINT"X is an ordinary number equal to ";X
30 PRINT" X% is an integer."
40 X%=X
50 PRINT"The value of X% is ";X%
60 Y%=7/5
70 PRINT"7/5 is ";Y%;" in integers!"

```

Fig. 3.19. Using integer variables. Any fractions will be rejected.

other type of variable. The QL allows you to carry out integer division, using the functions DIV and MOD. DIV means the whole-number result of an integer division. If you type:

```
PRINT 7 DIV 3
```

for example, you will see the result 2 appear. This is because $7 \div 3$ is 2 and a fraction, and integer division ignores fractions. MOD is used to find the remainder after an integer division. If you type:

```
PRINT 7 MOD 3
```

then the result you see this time is 1. This is the *remainder* after 7 has been divided by 3. DIV and MOD come into their own when you get involved in some more advanced programming methods that we have space for in this book.

Another action which is specially useful for mathematical work is defined functions. This is a way of making use of a mathematical action many times, but writing it just once into your program. Figure 3.20 shows a way of using a defined function which works. It's a very simple

```

10 CLS
20 PRINT"Enter three numbers, please"
30 INPUT A,B,C
40 PRINT"Sum is ";sum(A,B,C)
100 STOP
1000 DEFine FuNction sum(A,B,C)
1010 RETurn A+B+C
1030 END DEFine

```

Fig. 3.20. Introducing the DEFined Function (DEF FN).

example, and you would never use a defined function for anything so easy, but being easy makes it simpler to follow. Lines 20 and 30 ask you to input three numbers. Line 40 then prints a quantity called

'sum(A,B,C)'. Now this has no meaning unless it is defined, and it has to be defined by a line that starts with DEF FN. When you enter this line, the QL will print the full version, DEFine FuNction, and you have to follow this with the name that you have decided on. The next line following the DEF FN starts with RETURN, and shows what you expect the function to do. In this case, it's going to add the numbers that have been assigned to A, B, and C. This actually ends the work, and the next line, END DEF is not actually needed in this example. Figure 3.21 shows another example, which follows the manual example more closely. In this case, the definition is given early in the program, and a quantity 'side' is defined as being 'local'. Now a local quantity is one

```

10 DEFine FuNction hypot (a,b)
20 LOCAL side
30 side=SQRT(a^2+b^2)
40 RETURN side
50 END DEFine
60 CLS
70 INPUT"Two sides of a triangle are
? ";x,y
80 PRINT"The third is ";hypot(x,y)

```

Fig. 3.21. Another, more useful, example of defined function.

which is used only in a part of a program, and has no value anywhere else. In this case, we want 'side' to be the answer that is returned to the main program, and so line 40 allocates 'side' as a quantity which has to be returned. You can prove that this is local, however, because after you run the program, the command PRINT side will produce just an asterisk, no number. Another thing to look for is that lines 70 and 80 use variables 'x' and 'y' as the two sides of the triangle. The DEF FN part uses 'a' and 'b'. The point is that the DEF FN part tells the computer what to do with a pair of numbers, and it doesn't matter what they are called. This makes programming very much easier when you are getting past the beginner stage and starting to flex your muscles a bit.

Chapter Four

Repeating Yourself

One of the activities for which a computer is particularly well suited is repeating a set of instructions and every computer is well equipped with instructions that will cause repetition. The QL is no exception to this rule, and it is equipped with more of these 'repeat' commands than any other computer I know. We'll start with one of the simplest of these 'repeater' actions, REPEAT. Now I know that this is not the simplest possible repeater action, because there is an instruction GOTO which is simpler. The point is that GOTO is an out-of-date command which causes more trouble than it's worth. It has been abandoned by the two machines which use a really modern version of BASIC – and one of these is the QL. There's no need to use GOTO nowadays – after all, if you order yourself a new Rolls-Royce, you don't insist on having running-boards welded on because they had them in 1938, do you?

REPEAT means exactly what you would expect it to mean – repeat some action that is going to be specified. Normally a program is carried out by executing the instructions in ascending order of line number. In plain language that means starting at the lowest numbered line, working through the lines in order and ending at the highest numbered line. Using REPEAT can break this arrangement, so that a line or a set of lines will be carried out over and over again. To make this happen we need a line which contains the keyword REPEAT, and a name for the routine, then a line or set of lines that tells the machine what it is you want to do over and over again. You have to end the sequence with an END REPEAT, using the same 'name', which shows that this is the last part of the repeating operation.

Figure 4.1 shows an example of a very simple repetition or 'loop', as we call it. Line 10 contains the instruction, REPEAT action. In this line, 'action' is a 'filename' or reference name for what is to be repeated. You could use whatever name you liked, so long as it isn't a reserved word or the name of a variable whose value you will want to use later. When line 10 has been carried out, the program moves on to line 20,

```

10 REPEAT action
20 PRINT "This is the QL"
30 END REPEAT action

```

Fig. 4.1. A very simple loop. You can stop this by pressing the CTRL and SPACE keys.

which instructs it to print the phrase 'This is the QL'. In this simple example, that's all we ask it to do. Line 30 then ends the REPEAT action, using the same name of 'action'. If you try to omit the name, you will get an error message. Now there is nothing in this set of instructions which could cause the loop to stop repeating, and so it will cause the screen to fill with the words:

This is the QL

until you press the CTRL and spacebar keys to 'break the loop'. Any loop that appears to be running forever can normally be stopped by pressing these keys. This does stop the program running, but not completely. If you type CONTINUE, and then press ENTER, the program will take over from where it left off. We'll see later that this is very useful if you are chasing faults in a program. If you want to stop the program completely, so that you can record it or change it, then you have to type CLEAR, then press ENTER. It's important to use CLEAR after stopping a program in this way, if you intend to make changes. If you don't use CLEAR, you may find that the machine 'locks-up' on you later, when you least expect it. Notice, by the way, that when you use CTRL-space to stop an endless loop, you get a curious little error message. In my case, it was 'In line 20, not complete'. This is intended as a reminder to you to use either CONTINUE or CLEAR.

Now try a loop in which there is slightly more noticeable activity. Figure 4.2 shows a loop in which a different number is printed out each time the computer goes through the actions of the loop. We call this 'each pass through the loop'. Line 10 sets the value of the variable N at 10. Line 20 is the start of the REPEAT section, which contains two

```

10 CLS:N=10
20 REPEAT loop
30 PRINT N
40 N=N+1
50 END REPEAT loop

```

Fig. 4.2. A loop which carries out a countdown action very rapidly. You will also have to use the CTRL/STOP keys to stop this one.

lines of commands. The first of these is line 30, PRINT N, and the second is line 40, $N=N+1$. Line 50 contains the END REPEAT for this loop, so that the program will cause a very rapid count-up to appear on the screen. Once again, you'll have to use the CTRL and spacebar keys to stop it, and this gives you a chance to see how the program will carry on the next time that you use CONTINUE. As before, you have to type CLEAR if you want to end it all completely.

Now an unending loop like this is not exactly good to have, and REPEAT is a method of creating loops that we can use much more constructively. To do this, we need a way of stopping the loop when we want to. This will be when some condition has been fulfilled. You might, for example, want to stop a loop when a number variable reaches a value of 100, or when the name "LASTONE" is entered. The extra commands that you need to know about in order to do this are IF...THEN and EXIT.

IF has to be followed by a condition. You might use conditions like IF $N=20$, or IF NAME\$="LASTONE" for this purpose. After the condition, you can use the word THEN, and that's all you are allowed to use in this 'condition' line. The next lines that you use can contain other instructions. You might simply want the loop to stop when the condition is true. This can be programmed by using the word EXIT, which has to be followed by the 'name' of the REPEAT routine. As usual, the QL is very fussy about names following these commands, and about spaces between commands and names. You can, however, omit the THEN, so long as there is nothing else put in its place. As we'll see later, you can have several conditions between the IF part of the test and its end. The end of testing is marked by the words END IF. This allows you as many lines of tests as you like to use. Finally, the end of the REPEAT has to be marked in the usual way.

Now if all of that sounds rather complicated, take a look at the simple illustrations in Fig. 4.3. Lines 10 to 30 contain the instructions.

```

10 CLS
20 PRINT"Enter some names --"
30 PRINT"entering LASTONE will stop t
he loop."
40 REPEAT names
50 INPUT name$
60 IF name$="LASTONE" THEN
70 EXIT names
80 END IF
90 END REPEAT names
100 PRINT"END OF PROGRAM"
```

Fig. 4.3. Detecting an ending condition for a loop, using IF and EXIT.

You are allowed to enter names, and the program will stop repeating the entry process when you enter the name of LASTONE. How do we program this? We start in line 40 with the REPEAT loop, using 'names' as the label for the routine. What we have to do over and over again is to INPUT a name, so line 50 attends to that. A 'real' program would do something with the name, but this is just an example to get you used to the techniques. After the name has been typed and the ENTER key pressed, you have to test the name to find if it is LASTONE. This is done in line 60. What do we do if it *is* LASTONE? The answer is in line 70 – EXIT names. This means 'break out of the loop'. You don't have to type CLEAR when you break out of a loop in this way; the machine sees to it all automatically. There are no other conditions that we want to try, so line 80 is the END IF (no name here!). Finally, line 90 contains the END REPEAT names which signals the last part of the loop. Line 100 prints a message just to show when you have jumped out of the loop.

Now when you run this, you'll find that it ends the loop when you type 'LASTONE', but not for 'Lastone' or 'lastone'. This is quite easy to deal with, because you can add some OR's to your test. For example, if you alter line 60 so that it reads:

```
60 IF name$="LASTONE" OR name$="Lastone" OR
   name$="lastone" THEN
```

– you are making three conditions in place of one. If *any one* of these conditions is true, whatever follows THEN in the next lines will be carried out. You can also use the word AND to couple two conditions. For example, if you want a REPEAT loop to end when a count number has got to 100 AND a name of LASTONE is entered, you can program this as:

```
60 IF name$="LASTONE" AND N=100 THEN
```

(assuming that your program uses a variable N to make a count).

What ELSE?

IF...THEN forms a test which can be used apart from REPEAT, but which we use most often in connection with a REPEAT type of loop. There's another extension to IF...THEN, however. You can use the word ELSE to carry out another test and cause a different sort of action. An example makes this a lot clearer, so take a look at Fig. 4.4.

This is a simple heads-or-tails gamble, with no scoring. Lines 10 to 30

```

10 CLS
20 PRINT"HEADS OR TAILS"
30 PRINT"Type E to stop it"
40 REPEAT gamble
50 X=RND(1,2)
60 INPUT"Heads (H) or Tails (T)? ";side$
70 IF side$="E" THEN
80 EXIT gamble
90 END IF
100 IF side$="H" AND X=1 OR side$="T"
    AND X=2 THEN
110 PRINT"You win"
120 ELSE PRINT"You lose"
130 END IF
140 END REPEAT gamble
150 PRINT"Game over"

```

Fig. 4.4. Using ELSE to extend the action of IF.

set things up as usual, while line 40 starts the main loop of repeated actions. Line 50 is the important gambling line. RND means 'select at random', and when it is followed by numbers in brackets, it means that the machine has to pick a whole number at random, lying between (and including) these limits. In this example, the limits are 1 and 2, so that each time line 50 runs, either a 1 or a 2 will be assigned to variable name X. Line 60 then asks you to type H or T for heads or tails. Whatever you type is assigned to a variable called side\$. Now we have to test side\$. The first test is to find if we want to break out of the loop. This is done by typing 'E' (on its edge!), and lines 70 to 90 make this test. We use EXIT here because we want the program to stop if E is typed. The other tests are made in line 100 to 130. Line 100 checks for success. If we take X=1 as meaning 'heads' and X=2 as meaning 'tails', then this line detects success. Line 110 will then print 'You win', and the loop repeats. Line 120, however, checks failure. If your guess H or T, did not agree with the selection of X, the ELSE line operates. This prints 'You lose', and once again, allows the loop to continue. Only if the 'E' has been typed does the game end with the message in line 150.

The importance of ELSE is that you can have an option. If a test succeeds, something can be done (a message printed, perhaps), and with the use of ELSE, another action (a different message, perhaps) can be taken if the test fails. Later on we'll look at how we can program for a large number of tests. For the moment, it's time to look at another type of loop that deals with numbers rather than with strings.

The FOR...END FOR loop

There is an alternative to the REPEAT type of loop when you want actions to be repeated a number of times. I really mean *number*, in the sense that the condition which ends the loop is not usually a string being equal to another string, but a number count finishing. This type of loop, on other machines, is called the FOR...NEXT loop, but on the QL it could also be named the FOR...END FOR loop. As the name suggests, this makes use of two new instructions, FOR and NEXT or END FOR. The instructions that are repeated are the instructions that are placed between FOR and NEXT, or between FOR and END FOR. The QL allows you to make use of FOR in two different ways, however. In the simple form, you don't need anything (like NEXT or END FOR) to mark the last action in the loop! This is possible only if all the instructions of the loop are in one line. Figure 4.5 illustrates a very simple example of this type of FOR loop in action. The line which

```
10 CLS
20 FOR N=1 TO 10:PRINT"QL BASIC RULES
, OK?"
```

Fig. 4.5. A simple one line FOR loop that needs no NEXT or END FOR.

contains FOR must also include a number variable which is used for counting, and numbers which control the start of the count and its end. In the example, N is the counter variable, and its limit numbers are 1 and 10. The action is a PRINT, and there is nothing else to do. Because the FOR and its action are both in one line, no NEXT or END FOR is needed. The effect of this program, then, will be to print QL BASIC RULES, OK? ten times. At the first pass through the loop, the value of N is set to 1, and the phrase is printed. When the end of the line is encountered, the computer increments the value of N, from 1 to 2 in this case. It then checks to see if this value exceeds the limit of 10 that has been set. If it doesn't, then the PRINT instruction is repeated, and this will continue until the value of N exceeds 10 – we'll look at that point later. The effect in this example is to cause ten repetitions.

This, however, is a very simple sort of loop. The QL also provides for loops that contain several lines of instructions between the start and the end. Before we start to go into such complications, let's see how our simple loop would look in the extended form. Figure 4.6 demonstrates this, using exactly the same actions. This time, the FOR N=1 TO 10 is in a line of its own, the PRINT line follows, and line 40 marks the end of the loop. It's very important that the END FOR is followed by the

```

10 CLS
20 FOR N=1 TO 10
30 PRINT"QL BASIC RULES, OK?"
40 END FOR N

```

Fig. 4.6. Using the FOR...END FOR loop for a counted number of repetitions.

name of the variable that you have used for counting – you'll get a 'bad line' error message if you omit this name and try to enter the line without it. Having tried that, edit line 40 (see Appendix A on editing if you haven't tried this yet), and replace END FOR N with NEXT N. You'll find that the loop runs in exactly the same way.

So why do we have two ways of ending the loop? The answer is that this allows for more choice. Suppose, for example, that you want your loop to contain an alternative message, or to stop with some other condition. Both of these possibilities are catered for in QL BASIC, as Fig. 4.7 shows. This looks like the same sort of count-up loop, but a

```

10 CLS
20 FOR N=1 TO 10
30 J=RND(1,10)
40 IF J=5 THEN EXIT N
50 PRINT"QL BASIC RULES, OK?"
60 NEXT N
70 PRINT"WE GOT TEN IN THIS TIME"
80 END FOR N
90 PRINT "THAT'S ENOUGH OF THAT"

```

Fig. 4.7. Using a test that allows the program to leave a loop before the full number of repetitions has been carried out.

random number is generated in line 30. This number must lie between 1 and 10, because of the use of RND (1,10), and line 40 uses $J=5$ as a condition that will stop the loop. Now in other machines, stopping a loop in this way can tie the computer in knots, but the QL handles it all smoothly. You use EXIT as before, following the condition (IF $J=5$), remembering that the 'name' of the variable, N, must follow EXIT. The loop will stop, then, if J happens to be 5 at any stage. Line 50 is the main loop action, and line 60 contains the NEXT. Now if the loop ends before the count of ten, because the random number line has picked a five, the program will never get to line 70. Line 70, following NEXT, is carried out only after ten repetitions of the loop. Because it comes *before* the END FOR N line, line 80, however, it can't be carried out if the loop has ended early because of J being 5. In this case, END FOR N marks out the end of *all* loop actions. In older versions of BASIC, this

sort of thing needs rather complicated looking lines of programming; in QL BASIC it is beautifully simple.

You don't have to confine this loop action to single loops either. Figure 4.8 shows an example of what we call 'nested loops', meaning that one loop is contained completely inside another one. When loops

```

10 CLS
20 FOR N=1 TO 10
30 PRINT"Count is ";N
40 FOR J=1 TO 500
50 NEXT J:CLS
60 END FOR N
70 PRINT"End of count"

```

Fig. 4.8. A program that uses nested loops, with one loop inside another.

are nested in this way, we can describe the loops as inner and outer. The outer loop starts in line 20, using variable N which goes from 1 to 10 in value. Line 30 is part of this outer loop, printing the value that the counter variable N has reached. Lines 40 and 50, however, contain another complete loop. This must make use of a different variable name, and it must start and finish again before the end of the outer loop. We have used variable J, and we have put nothing between the FOR part and the NEXT J part to be carried out. All that this loop does, then, is to waste time, making sure that there is some measurable time between the actions in the main loop. There is no special reason for using NEXT J rather than END FOR J in this loop, except that it makes it easier to see which loop is being ended. The last action of the main loop is clearing the screen following the NEXT J in line 50. The overall effect, then, is to show a count-up on the screen, slowly enough for you to see the changes, and wiping the screen clear each time. In this example we have used END FOR to indicate the end of the main loop. Whether you do use END FOR or NEXT, you must be absolutely sure that you have put the correct variable names following each END FOR or NEXT. If you use a variable name (for example, K) that you haven't assigned, you will get message like 'At line 50 not found'. If you mix up the variables, and make line 50 contain NEXT N, then the count will be rather fast.

Even at this stage it's possible to see how useful this FOR...END FOR loop can be, but there's more to come. To start with, the loops that we have looked at so far count upwards, incrementing the number variable. We don't always want this, and we can add the instruction

word STEP to the end of the FOR line to alter this change of variable value. We could, for example, use a line like:

```
FOR N=1 TO 9 STEP 2
```

which would cause the values of N to change in the sequence 1,3,5,7,9. When we don't type STEP, the loop will always use increments of 1.

Figure 4.9 illustrates an outer loop which has a step of -1, so that the count is downwards. N starts with a value of 10, and is decremented on

```
10 CLS
20 FOR N=10 TO 1 STEP -1
30 PRINT N;" seconds, and counting."
40 FOR j=1 TO 500:NEXT j
50 CLS
60 END FOR N
70 PRINT"BLASTOFF!!"
```

Fig. 4.9. A countdown program, making use of STEP.

each pass through the loop. Line 40 once again forms a time delay so that the countdown takes place at a civilised speed. You *can't* omit the NEXT J in this line – if you do, the delay simply doesn't work. A delay of this type is a particularly useful way of slowing a countdown. There's no such simple way to speed up a loop. A lot of computers will allow you to use integer variables in loops, so that you can start a loop:

```
FOR N%= 1 TO 10
```

but the QL rejects this as a 'bad line'. It's rather unfortunate, because the QL does not run loops particularly fast.

Every now and again, when we are using loops, we find that we need to use the value of N after the loop has finished. It's important to know what this will be, however, and Fig. 4.10 brings it home. This contains two loops, one counting up, the other counting down. At the end of

```
10 CLS
20 FOR N= 1 TO 5
30 PRINT N
40 NEXT N
50 PRINT"N is now ";N
60 FOR N=5 TO 1 STEP -1
70 PRINT N
80 NEXT N
90 PRINT"N is now ";N
```

Fig. 4.10. Finding the value of the loop variable after a loop action is completed.

each loop, the value of the counter variable is printed. This reveals that the value of N is 5 in line 50, after completing the FOR N = 1 to 5 loop, and is 1 in line 90 after completing the FOR N = 5 TO 1 STEP -1 loop. If you want to make use of the value of N, or whatever variable name you have selected to use, you will have to remember that it will have the final value that is specified in the loop command. *This is unusual*, because most other computers change N by one more step at the end of a loop. If you make use of programs that have been written for other machines, this is something that you will have to bear in mind.

One of the most valuable features of the FOR...END FOR (or FOR...NEXT) loop, however, is the way in which it can be used with number *variables* as well as just numbers. Figure 4.11 illustrates this in a simple way. The letters A, B and C are assigned as numbers in the usual way in line 20, but they are then used in a FOR...END FOR loop in line 30. The limits are set by A and B, and the step is obtained from an expression, B/C. The rule is that if you have *anything* that represents a number or can be worked out to give a number, then you can use it in a loop like this.

```
10 CLS
20 A=2:B=5:C=10
30 FOR N=A TO B STEP B/C
40 PRINT N
50 NEXT N
```

Fig. 4.11. A loop instruction that is formed with number variables.

Loops and decisions

It's time to see loops being used rather than just demonstrated. A simple application is in totalling numbers. The action that we want is that we enter numbers and the computer keeps a running total, adding each number to the total of the numbers so far. Now if we used a FOR...NEXT or END FOR type of loop here, we would only be able to enter a fixed amount of numbers, which means that we would have to count how many numbers were to be totalled each time. That's too much like hard work, and it would be a lot more convenient if we could just stop the action by signalling to the computer in some way, perhaps by entering a value like 0 or 999. A value like this is called a 'terminator', something that is obviously not one of the normal entries that we would use, but just a signal. For a number-totalling program, a terminator of 0 is very convenient, because if it gets added to the total it won't make any difference.

Figure 4.12, therefore, shows an example of this type of program in action. We can't use a FOR...END FOR loop, because we don't know in advance how many times we might want to go through the loop, so we have to use the REPEAT type of loop. The instructions appear first, and we then have to make the total variable 'total' equal to zero in line 30. When a program is RUN, each variable like this is automatically set to zero in any case, but it's good practice to set it in the program. The

```

10 CLS:AT 13,4:PRINT"TOTAL FINDER"
20 AT 1,6:PRINT"The program will tota
1 numbers for"\you. Enter a zero to
stop."
30 total=0
40 REPeat entry
50 INPUT"Number, please ";N
60 IF N=0 THEN EXIT entry
70 total=total + N
80 PRINT"Total so far is ";total
90 END REPeat entry
100 PRINT"Final total is ";total

```

Fig. 4.12. A number totalling program which repeats until a zero is entered.

reason is that if you start the program in any other way than RUN (like GOTO 10), the variable is *not* automatically set to zero. You might think that you would never do anything like that, but that's what they all say, your honour. Line 40 is the start of the REPEAT loop, and each time you type a number, in response to the request in line 50, the number that you type is assigned to variable N. This value of N is then tested in line 60. If the value is 0, then the EXIT command takes effect, and the next line that will be carried out will be the one following END REPEAT entry, line 100. If the value of N was not zero, it is added to the total in line 70, and line 80 prints the value of the total so far. Line 90 contains the END REPEAT command, so this is the end of the loop.

The effect, then, is that if the number which you have typed in line 50 was not a zero, line 90 will send the program back to repeat from line 50 onwards. This will continue until you do enter a zero. When this happens, the test in line 60 succeeds (N is zero), and the program jumps to the end of the loop, the line that follows the END REPEAT. This line (line 100) prints the final value of 'total', and the program ends. One peculiarity of the QL can ruin a program of this type, however, and you can see how for yourself. When the program starts, enter a number, say 2. The next time, don't press a number key, just the ENTER key. You'll see the error message 'At line 50 error in expression' appear, because

you have not entered a number. You will get the same effect if you enter a letter. You have to be careful with totalling programs because of this – if no key is pressed, then pressing ENTER will stop the program. We can avoid this by using a string variable N\$ in place of N, and testing it before we use it, but that's for when your programming has come on apace.

Now this type of loop allows you much more freedom than a FOR...END FOR loop, because you are not confined to the use of a number to decide how many repetitions you have. The key to it is the use of IF to make a decision – and that's what we need to look at more closely now.

Decisions, decisions

We can make a number of types of comparisons between number variables or numbers, and these are listed in Fig. 4.13. The

<i>Sign</i>	<i>Meaning</i>
=	Exact equality.
==	Almost equal (to one part in ten million).
>	Left-hand quantity greater than right-hand quantity.
<	Left-hand quantity less than right-hand quantity.
The signs can be combined as follows:	
<>	Quantities not equal.
>=	LHS greater than or equal to RHS.
<=	LHS less than or equal to RHS.

Note: When the < or > sign is combined with =, then the < or > sign must be used first. Using a combination like => will always cause a 'bad line' error message.

Fig. 4.13. The mathematical signs that are used for comparing numbers and number variables.

mathematical signs are used for convenience, and you have to remember which way round the 'greater than' and 'less than' signs have to be. It's important to note that the equals sign means 'identical to' when it is used in a test like this. If A is 3.9999999 and B is 4.0000000 then a test such as IF A = B will fail – A is not identical to B, even though it is close enough to be equal in our eyes. The important point here is that the numbers we see on the screen have been rounded, so that

PRINT A in the example above might give the result 4. The test, however, is made on the numbers which have not been rounded. You can get round this by using the test ' \approx ', which means 'almost equal to'. If the two numbers are equal for more than eight places of decimals, this comparison will come up as true. It's a useful point which no other machine had made use of.

Figure 4.14 shows another test - this time on string variables. The instructions are in lines 20 to 30; you are asked to type the Y or N key.

```

10 CLS
20 PRINT"Press the Y or N key"
30 PRINT"-Then press ENTER."
40 REPEAT query
50 INPUT A$
60 IF A$="Y" THEN
70 PRINT"That's YES"
80 EXIT query
90 ELSE IF A$="N"
100 PRINT"That's NO"
110 EXIT query
120 ELSE PRINT "Y OR N ONLY, PLEASE
130 END REPEAT query

```

Fig. 4.14. Testing string variables, in this example to find whether a reply is Y or N.

Line 40 starts a loop to get your answer, with an INPUT in line 50. You are expected to type Y or N and then press ENTER. The key that you have pressed has its value assigned to A\$, so that A\$ should be 'Y' or 'N'. Lines 60 to 120 then analyse this result. If the key that you pressed was 'Y', then line 70 prints 'That's YES', and the EXIT in line 80 takes effect. If A\$ is not 'Y', however, lines 70 and 80 are skipped, and the next test in line 90 is tried. If A\$ is 'N', the message in line 100 is printed, and the EXIT in line 110 operates. If, however, you have pressed neither 'Y' or 'N', all of these lines will be skipped, and the program moves to line 120. This patiently tells you that you have to use Y or N, and since the next line is the END REPEAT line, it sends the program back to the start of the loop at line 40 again.

The test in this example is for identity. Only if A\$ is absolutely identical to Y will the phrase 'That's YES' be printed. If you typed a space ahead of Y, or a space following it, or typed y in place of Y, then A\$ will *not* be identical, and the test fails. Failing means that A\$ is not identical to Y and everything that follows THEN in that line will be ignored. It's up to you to form these tests so that they behave in the way

that you want! The QL is one of a select group of computers that allows you to extend this IF...THEN test by the use of ELSE, and it offers, once again, much simpler programming than is available to owners of computers with 'Stone Age BASIC'.

Line 120 constitutes a 'mugtrap', a way of trapping mistakes. Very often when you have a choice of answers, you want to be sure that only certain replies are permitted. A mugtrap is a section of program that is intended to deal with an incorrect entry. A good mugtrap should show the user the error of his/her ways, and indicate what answer or answers might be more acceptable. This is very often important, because an incorrect entry in some types of program could cause the program to stop with an error message showing. For the skilled programmer (you, by the time you have finished this book!) this is just a minor annoyance, but for the inexperienced user it can cause a minor panic. A good program doesn't allow any entries that would cause the program to stop, because when a program stops, you *could* lose all the information that you had typed into it. You don't have to – you can type GOTO, followed by the correct line number, then press ENTER. This can allow you to get back to normal – if you know what line number you need. An inexperienced user would not know this, and unless you had designed the program yourself and had a printed listing, you might not know either. Mugtraps are our method of ensuring that mistakes do not stop a program.

Just to emphasise the sort of power that these simple instructions give you, Fig. 4.15 illustrates a very simple number-guessing game. Line 10 starts the loop which will be used when incorrect answers are given, and line 20 clears the screen, and the $X = \text{RND}(1, 10)$ step causes variable X to take a whole-number value that lies between 1 and 10. We can't predict what this value will be, because RND, as you already know, means 'select at random'. RND(1,10) picks numbers randomly enough for games purposes, but not quite randomly enough for serious statistical users. In lines 30 and 40, the instructions ask you to guess the size of the number, with the difference that you don't have to find it exactly. You enter your number at line 50, and the tests are made in lines 60, 90, and 120. If the number that you picked is identical to the random number, then you get the 'Spot on' message in line 70, and the program ends because of the EXIT in line 80. The less obvious test is in line 90. The expression $N - X$ is the difference between your guess, N, and the number X. If your guess is larger than the number, then $N - X$ is a positive number. If your guess is less than X, then $N - X$ is a negative number. The effect of ABS, however, is to make any number positive, so that if X were 5 and you guessed 6 or 4, then $\text{ABS}(N - X)$ would come

```

10 REPEAT number_guess
20 CLS:X=RND (1,10)
30 AT 10,2:PRINT"Guess the Number!"
40 AT 2,4:PRINT"If you get near, I'll
   tell you"
50 INPUT N
60 IF N=X THEN
70 PRINT "Spot on!"
80 EXIT number_guess
90 ELSE IF ABS(N-X)<3 THEN
100 PRINT"Close- it was ";X
110 EXIT number_guess
120 ELSE IF N<>X THEN
130 PRINT"Nowhere near! Try again"
140 FOR J=1 TO 1000:NEXT J
150 END IF
160 END REPEAT number_guess
170 PRINT"End of game"

```

Fig. 4.15. A simple number-guessing game which uses number comparisons.

to 1. If you get a difference of 1 or 2 (less than 3), the message in line 100 is printed. If you don't get anywhere near, the program repeats because of the final ELSE in line 120. This makes a test that looks unnecessary, because if the program reaches this line, N cannot possibly be equal to X. The reason for the comparison is that you must have an IF test following ELSE in order to make the ELSE part work. The effect is to print the message and then move to line 160 to repeat the process. It's very simple, but quite effective. You can see several flaws, however. Why should the program end after each successful attempt? Why not stay in the loop unless a zero is entered? How about devising a scoring system? You could score 3 for a 'Spot on' and 1 for a 'Close', for example. Try working on this one to incorporate your own ideas, because it's only by working on programs for yourself that you really learn what programming is all about.

Single key reply

So far, we have been putting in Y or N replies with the use of INPUT, which means pressing the key and then pressing ENTER. This has the advantage of giving you time for second thoughts, because you can delete what you have typed and type a new letter before you press ENTER. For snappier replies, however, there is an alternative in the

form of INKEY\$. INKEY\$ is an instruction that carries out a check of the keyboard to find if a key is pressed. This checking action is very fast, and normally the only way that we can make use of it is by placing the INKEY\$ instruction in a loop which will repeat until a key is pressed. Figure 4.16 shows such a loop. The INKEY\$ instruction will produce a string quantity when any key is pressed, so we assign INKEY\$ to a string variable, K\$. In this way, when any key is pressed, the quantity

```

10 CLS
20 PRINT"Press any key..."
30 REPEAT test
40 K$=INKEY$: IF K$<>"" THEN
50 EXIT test
55 END IF
60 END REPEAT test
70 PRINT"It was ";K$

```

Fig. 4.16. Using INKEY\$ to find when a key has been pressed. Some keys will cause the program to operate, but will not print anything on the screen.

KEYROW ()

↓ 7 6 5 4 3 2 1 0 CODES →	SHIFT	CTRL	ALT	X	V	/	N	,
	8	2	6	Q	E	O	T	U
	9	W	I	TAB	R	-	Y	
	L	3	H	1	A	P	D	J
	1	CAPS LOCK	K	S	F	=	G	;
	1	Z	.	C	B	£	M	~
	ENTER	←	↑	ESC	→	\	SPACE	↓
	F4	F1	5	F2	F3	F5	4	7
	1	2	4	8	16	32	64	128

Fig. 4.17. The KEYROW numbers. For each row number, each key returns a code which KEYROW detects.

that it represents will be assigned to K\$, and if K\$ is a 'blank string', meaning that no key was pressed, the line loops back to its start again. Note how we indicate a blank string by using two quotes with no space between them. By using the program of Fig. 4.16 you can see the effect of pressing different keys. A few keys, such as the spacebar, will produce no visible character on the screen in line 70, but will nevertheless allow the program to jump out of its loop in line 50. Some keys, such as the function keys on the left-hand side of the keyboard produce a diamond pattern when they are pressed, and others have no effect at all.

There's another, quite different, method of testing for a key being pressed. This uses the command word **KEYROW**, and it's based on the idea that the keys are divided into eight groups, numbered from 0 to 7. Now there are eight keys allocated to each group, and each key can be made to provide a code number. Figure 4.17 shows the row numbers, and the codes for the keys. If, for example, you picked row 3, then pressing the 'S' key gives the number 8, and no other key will give this number *in this row*. **KEYROW** therefore allows us to test, very simply, for one specific key, and arrange it so that no other key has any effect.

The program in Fig. 4.18 illustrates this use of **KEYROW**. The word has to be followed by a number, in brackets, which has to be the number of a row, 0 to 7. You can **PRINT KEYROW(X)** or assign it, or test it. In this example, we set up a loop which continually tests the value of **KEYROW(1)**. If this is equal to 64, then the loop is left, because you get the value of 64 in row 1 by pressing the spacebar. If any other key is pressed, the loop continues on its merry-go-round, because the condition has not been achieved. **KEYROW** is a very useful way of testing for a key, particularly in games, because *any* key can be detected, including the arrowed keys, and the F-keys.

```

10 CLS
20 PRINT "PRESS SPACEBAR TO PROCEED"
30 REPEAT LOOP
40 IF KEYROW(1)=64 THEN
50 EXIT LOOP
60 END IF
70 END REPEAT LOOP
80 PRINT "THAT'S THE SPACE!"

```

Fig. 4.18. Using **KEYROW** to detect one specific key, the spacebar in this example.

Chapter Five

Strings and Other Things

String functions

In Chapter 3, we took a fairly brief look at number functions. If numbers turn you on, that's fine, but string functions are in many ways more interesting. What makes them that way is that the really eye-catching and fascinating actions that the computer can carry out are so often done using string functions. What's a string function, then? As far as we are concerned, a string function is any action that we can carry out with strings. That definition doesn't exactly help you, I know, so let's go into more detail.

A string, as far as the QL is concerned, is a collection of characters which is represented by a string variable, a name which ends with the dollar sign. You can pack practically as many characters as you like into a QL string – many computers permit a maximum of 255 characters per string, but the QL allows you up to 32768, and that's a longer string than most of us will ever need. When a QL string consists of only number characters, it can be used just as if it were a number. Again, this is something that other computers cannot do. Like other computers, however, the QL stores its strings in a special way, making use of what is called ASCII code. The letters stand for American Standard Code for Information Interchange, and the ASCII (pronounced Askey) code is one that is used by most computers. Figure 5.1 shows a printout of the ASCII code numbers and the characters that they produce on my printer (Epson FX 80).

Each character is represented by a number, and the range of numbers is from 32 (the space) to 127. On the printer, 127 produces a black square, but on the QL screen, you'll see a symbol that looks like the Greek letter sigma. You can assign characters to a string variable by using the equality sign, with or without the use of LET. When you assign in this way, you need to use quotes around the characters. You can also assign using INPUT, when no quotes are needed, and

32	33	!	34	"	35	#	36	\$	
37	%	38	&	39	'	40	(41)
42	*	43	+	44	,	45	-	46	.
47	/	48	0	49	1	50	2	51	3
52	4	53	5	54	6	55	7	56	8
57	9	58	:	59	;	60	<	61	=
62	>	63	?	64	@	65	A	66	B
67	C	68	D	69	E	70	F	71	G
72	H	73	I	74	J	75	K	76	L
77	M	78	N	79	O	80	P	81	Q
82	R	83	S	84	T	85	U	86	V
87	W	88	X	89	Y	90	Z	91	[
92	\	93]	94	^	95	_	96	`
97	a	98	b	99	c	100	d	101	e
102	f	103	g	104	h	105	i	106	j
107	k	108	l	109	m	110	n	111	o
112	p	113	q	114	r	115	s	116	t
117	u	118	v	119	w	120	x	121	y
122	z	123	{	124		125	}	126	~
127	■								

Fig. 5.1. The standard ASCII code numbers.

by using READ...DATA, when quotes are once again needed.

Now all number variables are represented in a different type of coding, one that uses the same number of codes no matter whether the value of the variable is large or small. Because a string consists of a set of number codes in the memory of the computer, one code for each character, we can do things with strings that we cannot do with numbers. We can, for example, easily find how many characters are in a string. We can select some characters from a string, or we can change them or insert others. Actions such as these are the actions that we call 'string functions'.

LEN strikes again

One of these string function operations that I mentioned was finding out how many characters are contained in a string. Since a string can contain up to 32767 characters, an automatic method of counting them is rather useful, and LEN is that method. LEN has to be followed by the name of the string variable, within brackets, and the result of using LEN is always a number so that we can print it or assign it to a number variable. If you have graduated to the QL from a Spectrum, you will have to get used to typing brackets around these variable names when you use functions like LEN. If you have used other computers, it's less of a change. You don't, however, need to put a space between the 'N' of LEN and the opening bracket.

Figure 5.2 shows a simple example of LEN in use. Line 20 assigns a

```

10 CLS
20 A$="THE QL EVOLVES"
30 PRINT "There are ";LEN(A$);" characters in the phrase ";!A$
40 INPUT "Try something for yourself"\
k$
50 PRINT k$ "\"-has ";LEN(k$);" characters"
```

Fig. 5.2. Introducing LEN, a member of the string function family.

variable and line 30 tells you how many characters are in this variable. Note the word 'characters', it's not the same as 'letters'. Each space, full-stop, comma and so on counts as a character for the purposes of a string, because each one is represented by an ASCII code number. Lines 40 and 50 illustrate how you can find the length of a string which you have entered. This is something that is useful if you want to ensure that a name entered at the keyboard is not too long for the computer to use. You might, for example, have a program that places names in a form, with columns of restricted width, such as fifteen characters. If too long a name is entered, it could spill over into another column. You'll probably notice, if you have a long name or address, that when you get letters that have been computer-addressed, some part of the name or address may have been shortened. Now you know why, and how!

All this is hardly earth-shattering, but we can turn it to very good use, as Fig. 5.3 illustrates. This program uses LEN as part of a routine which will print a string called T\$ centred on a line. This is an extremely useful routine to use in your own programs, because its use can save you a lot

```

10 CLS
20 T$="The QL show in action"
30 col=(37-LEN(T$))/2
40 AT col,2:PRINT T$

```

Fig. 5.3. Using LEN to print titles centred.

of tedious counting when you write your programs. The principle is to use LEN to find out how many characters are present in the string T\$. This number is subtracted from 37, and the result is then divided by two. If the number of characters in the string is an even number, then the number 'col' will contain a .5, but this is completely ignored by AT when the string is printed. You can, incidentally, use 37 or 38. Whichever one you use, you will find that words are reasonably well centred – 38 works better with phrases which have an even number of letters, and 37 works better with phrases which have an odd number of letters. Yes, you could program that with a few IF...THEN...ELSE steps!

The whole simple process could be done in one line, but I have shown it in three lines so that you can see what the steps are. We can use a routine of this type to centre anything that has the name T\$. In the next chapter, we'll be looking at the idea of 'procedures', which allow you to type the set of instructions (for centring a title, for example) just once, and then use them for any string that you like.

By the left, slice!

The next group of string operations that we're going to look at are called *slicing operations*. The result of slicing a string is another string, a piece copied from the longer string. String slicing is a way of finding what letters or other characters are present at different places in a string. There's also a variation on this theme which allows you to insert new bits into a string.

All of that might not sound terribly interesting, so take a look at Fig. 5.4. The string A\$ is assigned in line 20, and another string b\$ is assigned in line 30. The two strings are then printed, with an underscore used as a spacer, in line 40. There is nothing special here. In line 60, however, a new string is assigned, and when you see it printed in line 70, it produces 'Q.L.'. Now how did this happen? It will be familiar stuff to ex-ZX users (but look out for differences, later), but to former users of other computers, it's not exactly straightforward. The key to it is the use of A\$(1) and b\$(1). The '1' refers to the position of characters in the

```

10 CLS
20 A$="Quick processing"
30 b$="Longer satisfaction"
40 PRINT A$; " _"; b$
50 PRINT:
60 C$=A$(1)&" "&b$(1)&" "
70 PRINT"    - "; C$

```

Fig. 5.4. Using the simplest string slicing action to obtain a single letter from each string.

string, numbered from the start onwards. A\$(1) is therefore the Q of 'Quick', and b\$(1) is the L of 'Longer'. If we had used 2 in place of 1, then we would have picked the letters 'u' and 'o' – not such an interesting message, though!

Copying one letter of a string into another string is not terribly useful, though it's handy if you want to extract a set of initials from a name. We can make further use of this process of 'slicing', however, by specifying two numbers. Of these, one is the position number for the first letter of the sample, and the other number represents the position of the last letter. Take a look at Fig. 5.5, which shows how you can extract more than one letter. The letters that are extracted are 1 to 3, 'Tel', simply by using A\$(1 TO 3). Note that there *must* be a space on each side of the 'TO'. Former ZX owners please note that you *can't* omit the 1. All these Spectrum programs that you used with commands like A\$(TO 3) will have to be rewritten! You will get the 'out of range' message if you omit the first number ahead of the 'TO'.

```

10 CLS
20 A$="Telephone number."
30 PRINT A$(1 TO 3)

```

Fig. 5.5. Extracting several letters, using TO.

All right, Jack?

String slicing isn't confined to copying a selected piece of the left-hand side of a string. We can also take a copy of characters from the right-hand side of a string. This particular facility isn't used quite so much as the left-hand copying one, but it's useful none the less. Figure 5.6 illustrates the use of the right-hand slice action to make use of the last word in a phrase. There are more serious uses than this. You can, for example, extract the last four figures from a string of numbers like 010-242-7015. I said a string of numbers deliberately, because something

```

10 CLS
20 A$="That old Black Magic"
30 PRINT"Q.L "& A$(16 TO)

```

Fig. 5.6. Using TO to extract letters from the right-hand side of a string.

like this has to be stored as a string variable rather than as a number. If you try to assign this to a number variable, you'll get a silly answer. Why? Because when you type $N = 010-242-7016$ then the computer assumes that you want to subtract 242 from 10 and 7016 from that result. The value for N is -7248 , which is not exactly what you had in mind! If you use $N\$="010-242-7016"$ then all is well. Note that you don't have to specify the number of the last character of the string when you use this command. You could, of course, find it by using `LEN`, but it isn't needed. When there is nothing following the TO in the brackets, the computer takes it that you want all the characters to the end of the string.

Now we can get quite a lot of interesting effects from these slicing methods. Take a look at Fig. 5.7 for an example, which does odd things with the letters of your name. The program prompts you to enter your

```

10 CLS
20 INPUT"Your name, please"\name$
30 N=LEN(name$)
40 FOR J=1 TO N
50 PRINT name$(1 TO J); "    ";name$(J
   TO )
60 NEXT J

```

Fig. 5.7. Slicing both left and right sides of a string.

name in line 20, and the name is assigned to `name$`. In line 30, we use `LEN` so that the number variable N contains the total number of characters in your name. This will include spaces and hyphens – nobody's likely to use asterisks and hashmarks! Line 40 starts a loop which uses the total number of characters as its end limit. Line 50 is the action line. When J is 1, line 50 prints the first letter on the left of your name on to the left-hand side of the screen, and the rest of your name a couple of spaces away. This depends, of course, on how many spaces you typed between the quotes in line 50. On the next pass through the loop, a new line is selected, and two letters are printed on the left side, with two fewer on the right side. This continues until the entire name is printed.

Middle of the road?

There's another variation on string slicing which is capable of much more than extracting from left or right. The instruction word is still TO, and it has to be used with two numbers, within brackets. As you would expect by now, the number which comes before the TO is the position of the starting letter, and the number which follows the TO is the ending letter of the piece that you want to slice. It's a lot easier to see in action than to describe, so try the program in Fig. 5.8. Line 20 assigns A\$ to a

```
10 CLS
20 A$="Reference No. PDQ123 #7146"
30 PRINT"Stores Ref. is "; A$(15 TO 2
0)
```

Fig. 5.8. Using TO to extract from any part of a string. This action can be controlled by variables.

phrase, and line 30 prints another phrase, making use of a set of reference letters and digits from the original phrase. It's a very simple example, but it shows TO being used to extract from the middle of a string, which is what it's all about. The number which comes before TO is the position number of the first letter you want in your slice, and the number which follows TO is the position number for the last letter.

This slicing business, however, can be a two-way trade, as Fig. 5.9 illustrates. Line 10 allocates a string variable to the word 'SAND-

```
10 A$="SANDSTONE"
20 PRINT A$
30 A$(8 TO 9)="RM"
40 PRINT A$
```

Fig. 5.9. Inserting letters into a string with TO.

STONE'. This is printed, and then line 30 inserts two letters into the string. This makes no difference to the length of the string, it simply means that letters 8 and 9 are 'RM' in place of 'NE'. When the string is printed again in line 40, it shows the change. What would happen if you tried to insert too much? Try it for yourself – make line 30 read:

```
30 A$(8 TO 10)="RMS"
```

and run this. As you'll see, you still get 'SANDSTORM', with no final 'S'. You cannot make a string *longer* by an insertion like this. Can you make it shorter? Type another line 30:

```
30 A$(5 TO 9)=" "
```

and run this. The word that is printed in line 40 is 'SAND', so the word *has* been shortened – but the string has not! If you type PRINT LEN(A\$), you'll get the result of 9, so that what has happened is that each letter following the 'D' of 'SAND' has been replaced by a space. This can cause a lot of problems if you are using the length of strings for anything important, like centring titles or placing strings into columns.

One of the features of all of these string slicing instructions is that we can use variable names or expressions in place of numbers. Figure 5.10 shows a more elaborate piece of slicing which uses expressions. It all

```

10 CLS
20 INPUT "Your name, please "; NM$
30 L=LEN(NM$):C=INT(L/2)
40 FOR N=1 TO C
50 AT 19-N,N:PRINT NM$(C-N+1 TO C+N)
60 NEXT N

```

Fig. 5.10. Making a letter pyramid to show the action of TO with a formula.

starts innocently enough in line 20 with a request for your name. Whatever you type is assigned to variable NM\$, and in line 30 a bit of mathematical juggling is carried out. How does it work? Suppose you type as your name DONALD. This has six letters, so in line 30 L is assigned to 6, and C is the whole number part of $L/2$ (equal to 3). Line 40 then starts a loop of 3 passes. In the first pass you print at position 18,1 – because $N=1$ and $19-N=18$. What you print is the name NM\$(3 TO 4). With $N=1$ and $C=3$, then $C-N+1$ is $3-1+1$, which is 3, and $C+N$ is $3+1$, which is 4. What you print, then, is the third and fourth letters of the name. On the next run through the loop, N is 2, $C-N+1$ is 2, and $C+N$ is 5. What is printed is NM\$(3 TO 5), which is ONAL. The loop goes on in this way, and the result is that you see on the screen a pyramid of letters formed from your name. It's quite impressive if you have a long name! If your name is short, try making up a longer one.

Important warning

Most varieties of BASIC on other machines make use of the commands LEFT\$, RIGHT\$ and MID\$. You *must not* use these commands on the QL. Unlike most incorrect commands, which are ignored, these caused my QL to 'crash' into an endless loop with a pattern at the bottom of the screen. I had to use the reset button to recover from this, and this caused the program to be lost. It is possible that later versions of the QL will have corrected this problem, but you will have to be careful when you are converting programs that were written for other machines.

More priceless characters

It's time now to look at some other types of string functions. If you hark back a few pages, you'll remember that we introduced the idea of ASCII code. This is the number code that is used to represent each of the characters that we can print on the screen. We can find out the code for any letter by using the function `CODE`, which is followed, within brackets, by a string character in quotes or a string variable (no quotes). The result of `CODE` is a number, the ASCII code number for that character. If you use `CODE("QL")`, then you'll get the code for the 'Q' only, because the action of `CODE` includes rejecting more than one character. Figure 5.11 shows this in action. String variable `A$` is

```
10 A$="QL Computing"
20 CLS:PRINT
30 FOR N=1 TO LEN(A$)
40 PRINT CODE(A$(N)); " ";
50 NEXT N
```

Fig. 5.11. Using CODE to find the ASCII code for letters.

assigned in line 10 and in line 30 a loop starts which will run through all the letters in `A$`. The letters are picked out one by one, using `A$(N)`, and the ASCII code for each letter is found with `CODE`. The space between quotes, along with the semicolons in line 40 make sure that the codes are all printed with a space between the numbers, and without taking a new line for each number. Simple, really.

`ASC` has an opposite function, `CHR$`. What follows `CHR$`, within brackets, has to be a code number, and the result is the character whose code number is given. The instruction `PRINT CHR$(65)`, for example, will cause the letter A to appear on the screen, because 65 is the ASCII code for the letter A. We can use this for coding messages. Every now and again, it's useful to be able to hide a message in a program so that it's not obvious to anyone who reads the listing. Using ASCII codes is not a particularly good way of hiding a message from a skilled programmer, but for non-skilled users it's good enough. Figure 5.12 illustrates this use. Lines 50 and 60 contain an `INKEY$` loop to make the program wait for you. When you press a key, the loop that starts in line 70 prints 12 characters on the screen. Each of these is read as an ASCII code from a list, using a `READ...DATA` instruction in the loop. The `PRINT CHR$(N)` in line 80 then converts the ASCII codes into characters and prints the characters, using a semicolon to keep the printing in a line. Try it! If you wanted to conceal the letters more thoroughly, you could use quantities like one quarter of each code

```

10 CLS:AT 2,2
20 PRINT"What's the meaning of QL?"
30 AT 2,4:RESTORE
40 PRINT"Press any letter for the answer"
50 K$=INKEY$: IF K$="" THEN
60 GO TO 50:END IF
70 FOR J=1 TO 12
80 READ N:PRINT CHR$(N);
90 NEXT J
100 STOP
110 DATA 81,117,97,110,116,117,109,32
,76,101,97,112

```

Fig. 5.12. Using ASCII codes to carry a coded message, and then using CHR\$ to obtain the character that corresponds to a code number.

number, or 5 times each code less 20, or anything else you like. These changed codes could be stored in the list, and the conversion back to ASCII codes made in the program. This will deter all but really persistent de-coders! This example, incidentally, illustrates the use of READ and DATA in a loop. We would normally use READ and DATA only for information that we particularly wanted to keep stored in a program like this. Another important point is the use of RESTORE in line 30. If this is not included, then the program will run once, and from then on you will get an error message 'End of file in 80'. This means that there is no more data to be read, because it was all read the first time. We looked at this point earlier in Chapter 3.

While we are on the subject of RESTORE, there's another twist to this instruction in the form of placing a line number following RESTORE. RESTORE with a line number means that the DATA list will start again from the beginning of *that line*. You must make sure, of course, that the line number which you have chosen is a data line! Take a look at Fig. 5.13. This offers a choice of data to be read by making use of RESTORE along with a number. When you pick a number, it is used in line 80 to carry out a RESTORE command which has a line number equal to one thousand times your selected number following it. When a is 2, for example, RESTORE 1000*a means 'start reading DATA at line 2000'. Each DATA line contains four items, so that when lines 90 to 100 are carried out, four items will be read from whichever line has been picked. It's a useful way of selecting from a number of lists which will be used each time the program is used.


```

10 CLS
20 PRINT"Which list do you want?"
30 REPEAT choice
40 INPUT"Number 1 to 3 please ";a
50 SELECT a=1 TO 3:EXIT choice
60 PRINT"Choose 1 to 3 only, please"
70 END REPEAT choice
80 RESTORE a*1000
90 FOR N=1 TO 4
100 READ X:PRINT X;" ";:NEXT N
110 STOP
1000 DATA 1,2,3,4
2000 DATA 2,4,6,8
3000 DATA 3,6,9,12

```

Fig. 5.13. Using RESTORE to select a data line.

The law about order

We saw earlier in Fig. 4.13, how numbers can be compared. We can also compare strings, using the ASCII codes as the basis for comparison. Two letters are identical if they have identical ASCII codes, so it's not difficult to see what the identity sign, = , means when we apply it to strings. If two long strings are identical, then they must contain the same letters in the same order. It's not so easy to see how we use the > and < signs until we think of ASCII codes. The ASCII code for A is 65, and the code for B is 66. In this sense, A is 'less than' B, because it has a smaller ASCII code. If we want to place letters into alphabetical order, then, we simply arrange them in order of ascending ASCII codes.

This process can be taken one stage further, though, to comparing complete words, character by character. Figure 5.14 illustrates this use of comparison using the = and > symbols. Line 20 assigns a nonsense word – it's just the first six letters on the top row of letter keys. Line 30 then asks you to type a word. The comparisons are then carried out in lines 40 to 70. If the word that you have typed, which is assigned to B\$, is identical to QWERTY, then the message in line 50 is printed, and the program ends. If QWERTY would come later in an index than your word, then line 70 is carried out. If, for example, you typed PERIPHERAL, then since Q comes after P in the alphabet and has an ASCII code that is greater than the code for P, your word B\$ scores lower than A\$, and line 70 swaps them round. This is done by assigning


```

10 CLS
20 A$="QWERTY"
30 PRINT:INPUT"Type a word ";B$
40 IF B$=A$ THEN
50 PRINT"Same as mine!":STOP
60 ELSE IF A$>B$ THEN
70 Q$=A$:A$=B$:B$=Q$
80 END IF
90 PRINT"Correct order is ";A$;" then
   ";B$
100 STOP

```

Fig. 5.14. Comparing words to decide on their alphabetical order.

a new string, Q\$ to A\$ (so that Q\$ = "QWERTY"), then assigning A\$ to B\$ (so A\$ = "PERIPHERAL"), then B\$ to Q\$ (so that B\$ = "QWERTY"). Line 90 will then print the words in the order A\$ and then B\$, which will be the correct alphabetical order. If the word that you typed comes later than QWERTY, for example, TAPE, then A\$ is not 'greater than' B\$, and the test in line 60 fails. No swap is made, and the order A\$, then B\$, is still correct. Note the important point, though, that words like QWERTZ and QWERTX will be put correctly into order – it's not just the first letter that counts.

Put it on the list

The variable names that we have used so far are useful, but there's a limit to their usefulness. Suppose, for example, that you had a program that allowed you to type in a large set of numbers. How would you go about assigning a different variable name to each item? Figure 5.15 illustrates this. Lines 10 to 40 generate an (imaginary) set of

```

10 CLS:DIM A(10)
20 FOR N=1 TO 10
30 A(N)=RND (10,100)
40 NEXT N
50 PRINT
60 AT 14,2:PRINT"Marks List"
70 PRINT:FOR N= 1 TO 10
80 AT 2,N+3:PRINT"Item ";N;" received
   ";A(N);" marks"
90 NEXT N

```

Fig. 5.15. An array of subscripted number variables. It's simpler than the name suggests!

examination marks. This is done simply to avoid the hard work of entering the real thing! The variable in line 30 is something new, though. It's called a *subscripted variable*, and the 'subscript' is the number that is represented by N. The name that we use has nothing to do with computing, it's a name that was used long before computers were around. How often do you make a list with the items numbered 1,2,3 ... and so on? These numbers 1,2,3 are a form of subscript number, put there simply so that you can identify different items. Similarly, by using variable names A(1), A(2), A(3) and so on, we can identify different items that have the common variable name of A. A member of this group like A(2) has its name pronounced as 'A-of-two'.

The usefulness of this method is that it allows us to use one single variable name for the complete list, picking out items simply by their identity numbers. Since the number can be a number-variable or an expression, this allows us to work with any item of the list. Figure 5.15 shows the list being constructed from the FOR...NEXT loop in lines 20 to 40. Each item is obtained by finding a random number between 1 and 10, and is then assigned to A(N). Ten of these 'marks' are assigned in this way, and then lines 60 to 90 print the list. It makes for much neater programming than you would have to use if you needed a separate variable name for each number.

So far, so good, but one point has been omitted so far. The first line contains the instruction DIM A(10). If you omit this, you will find that the program can't run, and you get the error message 'Bad name' whenever the computer meets A(N). The computer has to be prepared for the use of this type of variable name – the preparation consists of getting some memory ready to receive the data. When you use DIM (meaning 'dimension'), the memory is allocated for the array. A line such as DIM A(10) actually allows you up to *eleven* items in an array, in fact, because we can use A(0) if we like, but you must not attempt to use A(11) or any higher number. You will get the error message 'out of range' if you do so.

The important DIM instruction, then, consists of naming each variable that you will use for arrays, and following the name with the *maximum number*, within brackets, that you expect to use. You aren't forced to use this number, but you must not exceed it. If you do, and your program stops with an error message, you will have to change the DIM instruction and start again – which will be tough luck if you were typing in a list of 100 names! Note that you can dimension more than one variable in a DIM line, as Fig. 5.16 shows.

Figure 5.16 extends this use of arrays variables another step further. This time you are invited to type a name and a mark for each of ten

```

10 CLS: DIM A(10), N$(10,20)
20 AT 1,10: PRINT "Please enter names a
nd marks"
30 FOR N=1 TO 10
40 INPUT "Name "; N$(N)
50 INPUT "Mark "; A(N)
60 NEXT N
70 CLS: Total=0
80 AT 14,2: PRINT "MARKS LIST"
90 FOR N=1 TO 10
100 AT 2,N+3: PRINT N$(N);
110 AT 19,N+3: PRINT A(N)
120 Total=Total+A(N)
130 NEXT N
140 PRINT
150 PRINT "Average is "; Total/10

```

Fig. 5.16. Using strings in one array, and numbers in another.

items. When the list is complete, the screen is cleared and a variable called 'Total' is set to zero in line 70. The list is then printed neatly, and on each pass through the loop the total is counted up (in line 120) so that the average value can be printed at the end. The important point here is that it's not just numbers that we can keep in this list form. The correct name for the list is an array, and Fig. 5.16 uses both a string array (names) and a number array (marks).

There is, however, an important difference between a string array and a number array when you use the QL. A lot of other computers allow you to dimension a string array just as you dimension a number array. The QL, as you might expect by now, is different. For a string array like this, you need *two* numbers following the DIM. The first of these is, as before, the maximum item number that you will use — ten in this example. The second number is the maximum length of the string. In the example, this has been set to 20. If you enter any name that consists of more than 20 characters, then it will be chopped to size. As it is, a name as long as 18 characters would cause the name to overlap the marks column, so this restriction can be turned to good advantage. It does mean, though, that you have to plan your use of string arrays much more carefully on the QL than on most other machines.

Rows and columns

You can imagine an array as a list of items, one after the other, but there

is a variety of array which allows a different kind of list, called a *matrix*. A matrix is a list of groups or items, with all the items in a group related. We could think of a matrix as a set of rows and columns, with each group taking up a row, and the items of a group in separate columns. Take a look at Fig. 5.17 to see how this works. We use here a variable

```

10 CLS: DIM N$(3,2,10)
20 FOR N=1 TO 3
30 FOR J=1 TO 2
40 READ N$(N,J)
50 NEXT J: NEXT N
60 FOR N=1 TO 3
70 AT 5,N+3: PRINT N$(N,1);
80 AT 25,N+3: PRINT N$(N,2)
90 NEXT N
100 DATA "Horse", "Foal", "Cow", "Calf",
"Dog", "Puppy"

```

Fig. 5.17. Making a matrix of rows and columns. Note the DIM method.

N\$ which has three subscript numbers. The first number is the row number, the second is the column number, and the third is, as before, the maximum length of the string. We need two FOR...NEXT loops to read data into this matrix. This is carried out in lines 20 to 50. The items are then printed in columns by the loop in lines 60 to 90. In this loop, the variable N is used as the row number and we use the column numbers 1 and 2. The rows contain animal names, and the columns separate the different names that we use for adult and for young animals respectively. This example has used a *string matrix*, but a number matrix is also possible, and it needs no length dimension number.

Figure 5.18 shows a much more ambitious matrix program. The idea is to store sets of names and telephone numbers which are fed in by you in the course of the loop in lines 20 to 50. Once the matrix has been filled, you can pick an initial letter for a name, and ask the computer to print out the name and number that it has located. I've left out mugtraps just to keep this example reasonably short, but you would certainly need some sort of mugtraps, even if only in the form of a message like:

PRINT " SORRY, CAN'T FIND ";JS;" ENTRIES"

What's new, then? Line 90 is the one to watch. You are trying to identify the first letter of a name, and the name is in the form of an array variable. JS, remember, is a first letter that you have picked, and you want to find if A\$(1,1) or A\$(2,1) or A\$(3,1), etc., might start with this letter. To find

```

10 CLS: DIM A$(50,2,20)
20 FOR N=1 TO 50
30 AT 2,8: INPUT "Name "; A$(N,1)
40 AT 2,10: INPUT " Tel. No. "; A$(N,2)
50 CLS: NEXT N
60 CLS: AT 12,8: PRINT "LIST COMPLETE"
70 AT 2,10: INPUT "Pick an initial lett
er "; J$
80 FOR N= 1 TO 50
90 IF J$=A$(N,1)(1) THEN
100 PRINT "Name - "; A$(N,1)
110 PRINT "Number - "; A$(N,2)
120 END IF
130 NEXT N

```

Fig. 5.18. Using a name and number matrix for a simple telephone directory application.

the first letter of a string array item, you need the rather curious-looking statement `A$(N,1)(1)`. If you think about it, though, this is quite logical. To pick the first letter of a variable called `A$`, you would use `A$(1)`, and so to pick the first letter of a variable called `A$(N,1)`, you use `A$(N,1)(1)`. You could just as easily pick two letters by using `A$(N,1)(1 TO 2)`.

Loose ends

There is one string dodge that we haven't looked at so far, `DATE$`. If you type `PRINT DATE$`, then you will see a date and time appear on the screen. This is in the form:

1963 Apr 08 06:31:55

and that was the date I got when I typed the command. They were just trying to convince me that delivery had taken less than 28 days, I think! You can alter this date by making use of the `SDATE` command. `SDATE` has to be followed by numbers, separated by commas, to represent the year, month, day, hour, minute and second to which you want to set the date. For example, if you typed:

`SDATE 1984,6,14,21,12,20`

you will have set the date so that `DATE$` will give you:

1984 Jun 14 21:12:20

– though by the time you have typed `PRINT DATE$`, the time will have moved on by several seconds.

There is another variable called `DATE`, a number variable, which controls the printout that you get from `DATE$`, but `DAY$` is more useful for most users. `DAY$` gives the day printed as a three-letter word, like `Wed` or `Sat`. You can also adjust the time, as distinct from altering the whole date, by using `ADATE`, followed by a number. The number is the number of seconds by which you want to advance or retard the clock. `ADATE 36`, for example, will put the clock forward by 36 seconds, and `ADATE -45` will put the clock *back* by 45 seconds. The provisional manual stated that the clock would have a battery backup power supply, so that the clock would continue to run even when the computer was switched off. There was no trace of this happening on mine. If a battery backup is provided on later versions, then `ADATE` will be useful when the clocks are adjusted for Summer Time, because `ADATE 3600` will put the clock forward by one hour.

One rather useful and unusual string function is `INSTR`. This is used to find if one string is contained in another. It's used in the form:

```
X=A$ INSTR B$
```

to find if `A$` is contained in `B$`. If it is, then `X` is the position number of the first letter of `A$` that is found in `B$`. If `A$` is *not* contained in `B$`, then `X` is zero. `X` will always be zero if `A$` is longer than `B$`. You can, of course, use the form:

```
PRINT A$ INSTR B$
```

If you just want to see the number.

Figure 5.19 shows a simple example of this function in action. Lines 20 to 40 allocate names to strings, and lines 50 to 70 make the tests, so that you can see how they work out. When you use names which are

```
10 CLS
20 A$="ALBERT HALL"
30 b$="RICHARDSON, BERTRAM"
40 C$="SINCLAIR, I"
50 PRINT"IN A$, BERT IS LOCATED AT"; "
  BERT" INSTR A$
60 PRINT"IN B$, BERT IS LOCATED AT ";
  "BERT" INSTR b$
70 PRINT"IN C$, BERT IS LOCATED AT ";
  "BERT" INSTR C$
```

Fig. 5.19. Making use of `INSTR`.

enclosed in quotes, there doesn't have to be a space between the quote mark and the INSTR instruction, but if you use string variables, then a space *must* be left. Having this command allows you to store, for example, a list of names in the form of a long string. You can locate the position of a name by using INSTR, and if the name is not present, it can be added to the end of the string. This avoids all of the dimensioning problems of a string array, not to mention the fixed length of strings in an array. When your programming advances to the extent that you want to design your own programs for this type of use, INSTR is a useful instruction to keep in mind.

Finally, another warning. If you are converting any programs that have been written for other machines, beware of the STR\$ command. Like LEFT\$, RIGHT\$ and MID\$, this one sends the QL bananas. On mine, it filled the screen with rubbish, caused the printer to operate, spun both Microdrives, and generally caused chaos. When the operating system of the QL is fully developed, commands like this should simply be rejected with an error message.

Chapter Six

Further Procedures

Figure 4.14 introduced the idea of making a choice, by pressing a key. In that example, the choice of keys was limited, Y or N. A choice of two items, isn't exactly generous, and we can extend the choice by a program routine that is called a *menu*. A menu is a list of choices, usually of program actions. By picking one of these choices, we can cause a section of the program to be run. One way of making the choice is by numbering the menu items, and typing the number of the one that you want to use. Figure 6.1 shows what a typical menu of this type would look like on the screen. With a computer which was fitted with

MENU

1. Make a new list of names.
2. Add names to list.
3. Read names from list.
4. Delete names from list.
5. Select one name.
6. End program.

PLEASE CHOOSE BY NUMBER

Fig. 6.1. How a typical menu might appear on the screen.

'Stone Age BASIC', we could use a set of lines such as:

```
IF K = 1 THEN 1000
IF K = 2 THEN 2000
```

and so on. There is a much simpler method, however, which uses a new QL instruction, SELECT ON. There are two ways in which SELECT can be used, and we'll look at both of them closely, because this command is not available in other versions of BASIC that you might have used.

To start with, suppose we want to pick from four items by typing numbers 1 to 4 on the keyboard. The first thing that you have to ensure is that only the numbers 1 to 4 are accepted, not numbers like 0, 5, -10 and so on — in other words, a bit of mugtrapping. The second point is that the use of INPUT is a bit tedious, because you have to press ENTER after you have typed your number-key. We'll use INKEY\$, then, to get the reply (the number-key), and it only remains to check that you have chosen a number that is in the correct range. Now SELECT can be used for this, and in the example of Fig. 6.2, there are two loops devoted to getting a reply from the keyboard and to testing this reply. Since this is a much longer program than you have used

```

10 CLS:AT 17,2: PRINT"MENU"
20 AT 2,4:PRINT"Please select by numb
er 1 - 4"
30 REPeat choice
40 REPeat key
50 K$=INKEY$
60 IF K$<>" " THEN
70 EXIT key
80 END IF
90 END REPeat key
100 k=K$
110 SElect k=1 TO 4:EXIT choice
120 PRINT"Incorrect - choice is 1 to
4 only"
130 PRINT"Please try again"
140 END REPeat choice
150 SElect ON k
160 ON k=1
170 PRINT"THIS IS THE FIRST CHOICE"
180 REMark
190 ON k=2
200 PRINT"THIS IS THE SECOND CHOICE"
210 REMark
220 ON k =3
230 PRINT"THIS IS THE THIRD CHOICE"
240 REMark
250 ON k=4
260 PRINT"THIS IS THE FOURTH CHOICE"
270 REMark
280 END SElect
290 PRINT"End of program"

```

Fig. 6.2. Using SELECT in menu programming.

before, perhaps it would be a good idea to look at it in sections. The trouble with SuperBASIC is that it needs a lot more typing to get the results!

Concentrate first of all on lines 10 to 140. Lines 10 and 20 tell you what it's all about, and the action starts in line 30. Now there are two loops, one starting in line 30 and one in line 40. When you find a program with nested loops like this, the easiest way of finding your way around it is to start at the innermost loop. In this example, that's lines 40 to 90. These lines are just an INKEY\$ loop - which in ordinary BASIC, incidentally, is written in one line. Figure 6.3 shows this part of the program written in more old-fashioned BASIC. Since it works just

```

30 REPEAT choice
50 K$=INKEY$
60 IF K$="" THEN GO TO 50
100 k=K$
110 SELECT k=1 TO 4:EXIT choice
120 PRINT"Incorrect - choice is 1 to
4 only"
130 PRINT"Please try again"
140 END REPEAT choice

```

Fig. 6.3. A simpler way of using INKEY\$.

as well, and is a lot easier to understand, we might as well stick with it from now on. Getting back to the longer version in Fig. 6.2, we've seen it before. If you press a key while this loop is operating, then lines 60 and 70 cause the program to jump to line 100. What this loop does, then, is to keep running round until a key is pressed.

At line 100, the statement `k=K$` converts a number in the string form of `K$` into number variable form `k`. Other BASICs have to use `k=VAL(K$)` to do this, but on the other hand, they won't stop with an error message if you press a letter key! Line 110 is the `SELECT` line. By using `SELECT k=1 TO 4`, we mean that this part of the program will accept this range of values, and if the value is in this range, then the next part of the line, `EXIT choice`, will be carried out. If the key that you pressed was 1,2,3 or 4, then, the program jumps to line 150. What if it's not? Well, if you pressed a letter key, then the program stops with an error message, because of the `k=K$` step in line 100. If you pressed a number that is out of the range, like 5, the `SELECT` line skips the `EXIT` part, and jumps to line 120, which tells you where you went wrong. This leads to the `END REPEAT choice` in line 140, which loops back to line 30 again. The main snag with this otherwise reasonable selection program is that the program is upset by pressing a letter key. You can't

get around this by using `SELECT K$ = "1" TO "4"`, because the variable that is used with `SELECT` *must* be a number variable. When `INKEY$` is used to find a number key, then, it's better to test the key using `CODE` (`IF CODE(K$) > 52 THEN...`) rather than by using `SELECT`.

Now that we have (we hope) the variable `K` representing our choice in terms of the number 1 to 4, we can make the choice. Lines 150 to 290 do so. The principle is to start with `SELECT ON k`, which means – choose the piece of program which contains the correct value of `k`. These pieces are each identified by starting with `ON k =`, and each piece can consist of as many lines of program as you like to use. In the example, because I was desperately trying to keep it short, I have used only a `PRINT` for each selection. If you selected 1, for example, line 150 will lead to line 160, and each line between this and the `ON k = 2` (in line 190) will be executed. Each other choice is dealt with in the same way – I have used a `PRINT` line and a `REM` line only. When a section has completed its action, the program jumps to the line that follows `END SELECT`. In this case, it is line 290, and I have used it to mark the end of the program. This may sound obvious, by the way, but it's always a good idea to print a phrase like 'End of Program' when a program ends. A surprising number of programs don't, and you are left staring at the screen wondering what more is expected of you.

Sectional programming

A lot of programs consist of a title, some instructions, and then a menu. Depending on what menu choice you have made, some part of the program is run, and the program ends, or returns to the menu again. The use of `SELECT`, as we have now seen, can make it reasonably easy for you to write programs of this style, about which there's more in this chapter. The idea of having a program which consists of a lot of separate pieces is very important. Every version of `BASIC` contains commands which allow you to have pieces of programs, called *subroutines*, which can be called for and used as needed. The `QL`, along with the `BBC Micro` before it, is able to make use of a much more useful type of program-piece, called a *procedure*. For the rest of this chapter, then, we'll be looking at `QL` procedures.

To start with, what's a procedure? Somebody once said that he couldn't really describe an elephant, but had no trouble recognising one if it walked by. A procedure is also less easy to describe than to illustrate, and so I'll mix description and illustration together so that

you get a good picture of this very useful action. A procedure is a command to the computer to carry out some action, and then return for the next command. In this respect, it's like any other computer command. When you type `PRINT A`, then the computer carries out the action of selecting a new line on the screen, which may involve doing a scroll. It then finds the value of the variable called 'A'. It then prints this value on the left-hand side of the screen line, and returns, waiting for the next command. If the variable A hasn't been assigned, then an asterisk is printed instead.

Now this is quite a lot of action to be called up by one word, and that's what computing is all about. In this case, the command word by itself was not sufficient, because you had to add the 'A'. The 'A' is a *parameter of the command*, something we add to a command to make it complete. `PRINT` by itself prints a blank line, but `PRINT A` prints the value of A. Some commands can use more than one parameter. For example, the `AT` command for positioning the cursor needs two parameters, the number of columns across the screen and the number of lines down.

On the QL, it's possible to define a command word for yourself, and that's what a procedure is all about. There are two parts to it. At one place in the program, you have to have some lines which explain what this new command word means. Once these lines are in place, you can then use the command word anywhere else in your program. If the command needs parameters, these must be supplied in the correct order, otherwise you'll get the inevitable error message. I think it's time for an example.

Figure 6.4 shows a very simple procedure in use. Lines 10 to 50 define what the procedure must do. In line 10 we have to use `DEF PROC` (and

```

10 DEFine PROCedure central (A$)
20 start=(38 - LEN(A$))/2
30 CLS
40 AT start,1:PRINT A$
50 END DEFine
60 central "This is a title"

```

Fig. 6.4. How a procedure is defined and used.

the computer will print this as `DEFine PROCedure`) to indicate that this is the start of a procedure definition. Next, we have to give the procedure a name, one that we'll use as a command word to call it up. I'm going to use the word 'central' for this one, because its purpose is to print a phrase centred on the top line of a clear screen. Like a `PRINT` command, then, there will be a parameter, which will be whatever we

want to print. In the definition, any parameters *must* be enclosed in brackets. When we use the command word, however, the brackets are optional – we can use them or leave them out without affecting the way that the procedure works.

Having got the definition line over, we then have to specify what the procedure called ‘central’ does. Line 20 defines a variable ‘start’, which is the column’s number for an AT command. This has been obtained in the way that was described in Chapter 2, using the same formula. This is, incidentally, the formula for use with a TV rather than with a monitor. Line 30 clears the screen, and line 40 contains the AT instruction, and the PRINT A\$. Once again, this raises the point about a ‘local’ variable. The A\$ is local to this procedure. What that means in plain language is that A\$ has a meaning in lines 10 to 50, but not anywhere else. If you have a line elsewhere, PRINT A\$, then it prints only an asterisk, meaning that there is no such variable! You can also use the name A\$ for anything else that you like outside the PROCEDURE lines. The computer will not confuse the A\$ in the procedure with the A\$ elsewhere. Clever, isn’t it?

Line 50 shows where the definition ends, and in line 60, this new command of ‘central’ is tested. By typing ‘central’ followed by “This is a title”, you are going to call up the procedure and pass it a parameter. The parameter consists of a string – ‘This is a title’. In the parameter lines, 10 to 50, this will be allocated to A\$, just as if you have typed a line:

A\$=“This is a title”

and the result is that the procedure gets to work on this string. The screen is cleared, and the words are printed centred. Magic!

Stick around; there’s even better to come. In this example, the lines that define the procedure have been numbered 10 to 50, and they come before the ‘call’ in line 60. This doesn’t have to be the arrangement that you use. The DEFine PROCEDURE lines can be anywhere in the program, as long as they start with DEF PROC and end with END DEF. If you do a bit of editing of the example in Figure 6.4, so that the DEF PROC line becomes 100, and the END DEF line is 500, then the first line of the program becomes line 60. This calls for a procedure which you might think had not been defined. There’s no problem about this, because the computer is arranged to look for a definition when it encounters a strange command word. It doesn’t matter whether the definition lines are at the start of a program or at the end or in the middle, the computer will find them. If it can’t, then you will get an error message.

Proceeding your own way

You can get a lot of enjoyment from your QL computer when you use it to enter programs from cartridges that you have bought. You can obtain even more enjoyment from typing in programs that you have seen printed in magazines. Even more rewarding is modifying one of these programs so that it behaves in a rather different way, making it do what suits you. The pinnacle of satisfaction, as far as computing is concerned, however, is achieved when you design your own programs. These don't have to be masterpieces. Just to have decided what you want, written it as a program, entered it and made it work is enough. It's 100% your own work, and you'll enjoy it all the more for that. After all, buying a computer and not programming it yourself is like buying a BMW and getting someone else to drive it for you.

Now I can't tell in advance what your interests in programs might be. Some readers might want to design programs that will keep tabs on a stamp collection, a record collection, a set of notes on food preparation or the technical details of vintage steam locomotives. Programs of this type are called 'database' programs, because they need a lot of data items to be typed in and recorded. On the other hand, you might be interested in games, colour patterns, drawings, sound, or other programs that require shapes to move across the screen. Programs of that type need instructions that we shall look at in quite a lot of detail in Chapters 8 to 10. What we are going to look at in this section is the database type of program, because it's designed in a way that can be used for all types of programs. Once you can design simple programs of this type, you can progress, using the same methods, to designing your own graphics and sound programs. Remember, though, that most of the very fast moving or elaborate graphics programs that you see are not written in BASIC. The reason is that BASIC is too slow to allow fast movement, or the control of lots of moving objects. These arcade-type programs, like the four business-type programs that come with your QL, are written in *machine code*, a set of number-coded instructions direct to the 68008 microprocessor that is the heart of the computer. This bypasses BASIC altogether, and is very much more difficult. If you learn how to design programs in BASIC, however, you will be able to learn machine code later. All you need is experience – a lot of it.

Two points are important here. One is that experience counts in this design business. If you make your first efforts at design as simple as possible, you'll learn much more from them. That's because you're more likely to succeed with a simple program first time round. You'll

learn more from designing a simple program that works than from an elaborate program that never seems to do what it should. The second point is that program design has to start with the computer switched off, preferably in another room! The reason is that program design needs planning, and you can't plan properly when you have temptation in the shape of a keyboard in front of you. Get away from it!

Put it on paper

We start, then, with a pad of paper. For myself, I use a 'student's pad' of A4 which is punched so that I can put sheets into a file. This way, I can keep the sheets tidy, and add to them as I need. I can also throw away any sheets I don't need, which is just as important. Yes, I said sheets! Even a very simple program is probably going to need more than one sheet of paper for its design. If you then go in for more elaborate programs, you may easily find yourself with a couple of dozen sheets of planning and of listing before you get to the keyboard. Just to make the exercise more interesting, I'll take an example of a program, and design it as we go. This will be a very simple program, but it will illustrate all the skills that you need.

Start, then, by writing down what you expect the program to do. You might think that you don't need to do this, because you know what you want, but you'd be surprised. There's an old saying about not being able to see the wood for the trees, and it applies very forcefully to designing programs. If you don't write down what you expect a program to do, it's odds on that the program will never do it! The reason is that you get so involved in details when you start writing the lines of BASIC that it's astonishingly easy to forget what it's all for. If you write it down, you'll have a goal to aim for, and that's as important in program design as it is in life. Don't just dash down a few words. Take some time about it, and consider what you want the program to be able to do. If you don't know, you can't program it! What is even more important is that this action of writing down what you expect a program to do gives you a chance to design a properly structured program. Structured in this sense means that the program is put together in a way that is a logical sequence, so that it is easy to add to, change, or redesign. If you learn to program in this way, your programs will be easy to understand, take less time to get working, and will be easy to extend so that they do more than you intended at first. The commands of the QL make it particularly easy to structure your programs properly.

As an example, take a look at Figure 6.5. This shows a program outline plan for a simple game. The aim of the game is to become familiar with the names of animals and their young. The program plan shows what I expect of this game. It must present the name of an animal, picked at random, on the screen, and then ask what the name of its young is. A little bit more thought produces some additional points.

Aims:

1. Present the name of an animal on the screen. Name should be picked at random from a list.
 2. Ask what its young is called.
 3. Reply must be correctly spelled, with a capital letter.
 4. User should not be able to read answer easily from listing.
 5. Give one point for each correct answer.
 6. Allow two chances at each question.
 7. Keep a tally of the number of attempts (not counting a second chance).
 8. Present the score as the number of correct answers out of the number of attempts.
-

Fig. 6.5. A program outline plan. This is your starter!

The name of the young animal will have to be correctly spelled. A little bit of trickery will be needed to prevent the user (son, daughter, brother, or sister) from finding the answers by typing LIST and looking for the DATA lines. Every game must have some sort of scoring system, so we allow one point for each correct answer. Since spelling is important, perhaps we should allow more than one try at each question. Finally, we should keep track of the number of attempts and the number of correct answers, and present this as the score at the end of each game. Now this is about as much detail as we need, unless we want to make the game more elaborate. For a first effort, this is quite enough. How do we start the design from this point on?

The answer is to design the program in the way that an artist paints a picture or an architect designs a house. That means designing the outlines first, and the details later. The outlines of this program are the steps that make up the sequence of actions. We shall, for example, want to have a title displayed, give the user time to read this, and then show instructions. There's little doubt that we shall want to do things like assign variable names, dimension arrays, and other such preparation. We then need to play the game. The next thing is to find the score, and then ask the user if another game is wanted. Yes, you have to put it all down on paper! Figure 6.6 shows what this might look like at this stage.

-
1. Print title, then instructions.
 2. Set up variables, read data, etc.
 3. Repeat – until ended –
 4. Pick random number;
 5. Play game;
 6. Find score, print score;
 7. Ask if another animal name is wanted;
 8. Until user does not reply 'Y'.
 9. Indicate that game is over.
-

Fig. 6.6. The next stage in expanding the outline.

Foundation stones

Now, at last, we can start writing a chunk of program. This will just be a foundation, though. What you must avoid at all costs is filling pages with BASIC lines at this stage. As any builder will tell you, the foundation counts for a lot. Get it right, and you have decided how good the rest of the structure will be. The main thing you have to avoid now is building a wall before the foundation is complete!

Figure 6.7 shows what you should aim for at this stage. There are only fourteen lines of program here, and that's as much as you want. This is a foundation, remember, not the Empire State Building! It's also a program that is being developed, so we shall not worry about using odd line numbers, because we can renumber it all later. Another point

```
10 CLS: CLEAR: RESTORE
20 title
30 instructions
40 setup
50 REPEAT play
55 pick
60 game
70 score
80 askmore
90 IF answer$="N" OR answer$="n" THEN
100 EXIT play
105 END IF
110 END REPEAT play
120 PRINT "End of game"
```

Fig. 6.7. A 'core' or 'foundation' program for the example.

is that we shall pick names for the PROCedures which remind us of what they are supposed to do.

Let's get back to the program itself. As you can see, it consists of a set of procedures that we haven't written yet. That's intentional. What we want at this point, remember, is foundations. The program follows the plan of Fig. 6.6 exactly, and the only part that is not committed as part of a PROC is the IF in line 90. What we shall do is to write a subroutine which will use INKEY\$ to look for a 'Y' or 'y' being pressed, and line 90 deals with the answer. What's the question? Why, it's the 'Do you want another game?' step that we planned for earlier.

Take a good long look at this fourteen-line piece of program, because it's important. The use of PROCedures means that we can check this program easily – there isn't much to go wrong with it. We can now decide in what order we are going to write the PROCedures. The wrong order, in practically every example, is the order in which they appear. Always write the title and instructions last, because they are the least important to you at this stage. In any case, if you write them too early, it's odds on that you will have some bright ideas about improving the game soon enough, and you will have to write the instructions all over again. A good idea at this stage is to write a line such as:

```
11 GO TO 40
```

which will cause the program to skip over the title and instructions. This saves a lot of time when you are testing the program, because you don't have the delay of printing the title and instructions each time you run it. Another point here is one which I hope was a peculiarity of my early QL. When I wrote the instructions in the form of one long PRINT instruction, using the forward slash sign to force new lines, the program would not record correctly. I lost several copies of the program in this way before finding what was causing the trouble. It's *always* a good idea to save programs in stages, saving under a new filename each time you add some lines. In this way, if you find that the memory of the QL has become scrambled, something that has happened to me too often for comfort, you may be able to load one of the earlier versions of the program, and avoid having to retype all of it.

The next step is to get to the keyboard (at last, at last!) and enter this core program. You can't test it until the PROC lines are written, but the next step is to record this core program and then keep adding to the core. If you have the core recorded, then you can load this into your QL, add one of the procedures, and then test. When you are satisfied that it works, you can record the whole lot with another filename. Next time you want to add a PROCEDURE, you start with this version, and so

on. This way, you keep tapes of a steadily-growing program, with each stage tested and known to work. Again, this is important. Very often, testing takes longer than you expect, and it can be a very tedious job when you have a long program to work with. By testing each PROCEDURE as you go, you know that you can have confidence in the earlier parts of the program, and you can concentrate on errors in the new sections. You can run a PROCEDURE, remember, simply by typing its name and pressing ENTER.

How to proceed

The next thing we have to do is to design the PROCEDURES. Now some of these may not need much designing. Take, for example, the PROCEDURE that is called 'askmore'. This is just our familiar INKEY\$ routine, along with a bit of PRINT, so we can deal with it right away. Figure 6.8 shows the form it might take. The PROCEDURE is straightforward, and that's why we can deal with it right away! Type it in, and now test the core program with this PROCEDURE in place.

```

1000 DEFine PROCEDURE askmore
1005 PRINT:PRINT"Do you want to try a
nother one (Y/N)?"
1010 REPeat query
1020 K$=INKEY$
1030 IF K$<>" " THEN
1040 answer$=K$
1050 EXIT query
1060 END IF
1070 END REPeat query
1080 END DEFine

```

Fig. 6.8. The 'askmore' PROCEDURE.

Now we come to what you might think is the hardest part of the job—the PROCEDURES which are enclosed in the 'play' loop. In fact, you don't have to learn anything new to do this. The main PROCEDURES are designed in exactly the same way as we designed the core program. That means we have to write down what we expect them to do, and then arrange the steps that will carry out the action. If there's anything that seems to need more thought, we can relegate it to a PROCEDURE to be dealt with later.

As an example, take a look at Fig. 6.9. This is a plan for the 'game' PROCEDURE, which also includes information that we shall need for the

1. Print name of animal on screen, ask what young is called.
2. Keep the list of animals as a string array.
3. Keep answers as ASCII codes in DATA lines, one line for each answer.
4. Use variables 'try' for each question count, 'mark' for score, 'Attempt' for number of attempts at one question.
5. Use Q\$ as name for array of animals.

Fig. 6.9. Planning the 'game' PROCEDURE.

setting-up steps in the 'pick' PROCEDURE. Before the 'game' PROCEDURE starts, we have to set a variable called 'Attempt' to zero. This is going to be used to decide whether or not the user gets another shot at a question. We also need to pick a number at random. These steps are taken care of by a PROCEDURE called 'pick'. This has to be separated from 'game', because if the user gets the answer wrong on a first attempt, you want to allow another try at the same animal, not to choose another one.

The first item in the 'game' PROCEDURE is to print the name of an animal on the screen, and ask for the name of its young. This will have been selected by the random number passed on from PROCEDURE 'pick'. This is most easily done by keeping the names of the animals in an array. This has several advantages. One of them is that it's beautifully easy to select one at random if we do this. The other is that it also makes it easy to match the answers to the questions. If the questions are items of an array whose subscript numbers are 1 to 10, then we can place the answers in DATA lines, one set of numbers in each data line, and in the same order, so that we can use RESTORE to pick the answer.

The next point is the result of a bit of thought. We wanted, you remember, to be sure that some smart user would not cheat by looking up the answers in the DATA lines. The simplest deterrent is to make the answers in the form of ASCII codes. It won't deter the more skilled, but it will do for starters. I've decided to put one answer in each DATA line in the form of a string of ASCII codes, with the first number in each line equal to the number of letters in the name. This way, we can RESTORE to the correct line, read the number of letters, and then start a FOR...NEXT loop which will read that number of codes. Each code can be converted to its character equivalent, using CHR\$, and added to the answer string, A\$.

The other thing that the plan settles is the names that we shall use for variables. It always helps if we can use names that remind us of what the variables are supposed to represent. In this case, using 'mark' for the

score and 'try' for the number of tries looks self-explanatory. We have used 'mark', because 'score' is the name of a PROCedure – you can't use a PROCedure name as a variable name. The third one, 'Attempt' is one that we shall use to count how many times one question is attempted. Finally, we decide on a name for the array that will hold the animal names – Q\$.

Proceed to play

Figure 6.10 shows the 'pick' PROCedure, and Fig. 6.11 shows what I've ended up with for 'game' as a result of the plan in Fig. 6.9. The steps are to pick a random number, use it to print an animal name, and then find the answer. That's all, because the checking of the answer and the scoring is dealt with by another PROCedure. Always try to split up the program as much as possible, so that you don't have to write huge chunks at a time.

```
1090 DEFine PROCedure pick
1095 Attempt=0:v=RND(1,10)
1097 END DEFine
```

Fig. 6.10. The 'pick' PROCedure.

```
1100 DEFine PROCedure game
1120 CLS:AT 2,2:PRINT "The animal is
";Q$(v)
1130 AT 2,4:PRINT"The young is called
";
1140 INPUT X$:try=try+1
1150 A$="":RESTORE 6000+v
1160 READ size: FOR J=1 TO size
1170 READ num:A$=A$ & CHR$(num)
1175 NEXT J
1180 END DEFine
```

Fig. 6.11. The program lines for the 'game' PROCedure.

We start the PROCedure at line 1100 and in line 1120 we print the name of the animal that corresponds to the random number, and ask for an answer, the young of that animal. The last section of line 1140 counts the number of tries. This is the logical place to put this step, because we want to make the count each time there is an answer. Now we set A\$ to a blank. This has to be done, because if you didn't, each answer would be tacked on to the end of the previous one! By using

RESTORE 6000+v, we find the DATA line that contains the correct answer. If, for example, v is 3, then the question name is in string Q\$(3), and the answer will be on line 6003. We read the first number in this line, 'size'. This is the number of characters in the answer. The loop in lines 1160 to 1175 reads in the numbers, converts them to characters, and adds them to A\$ to form the answer. That's it. All we have to do now is to compare this answer with the one that was INPUT in line 1140. This part of the work is dealt with by the 'score' subroutine.

Keeping the score

With the 'game' PROCedure safely on Microdrive, we can think now about the 'score' PROCedure. As usual, it has to be planned, and Fig. 6.12 shows the plan. Each time that there is a correct answer, the

-
1. For a correct answer, increment 'mark'.
 2. For an incorrect answer, with Attempt=0, allow another shot, and decrement 'try'.
 3. For second incorrect answer, with Attempt=1, pass on to the next question, and make Attempt=0 again.
 4. Sound different BEEP notes in each case (more about this in Chapter 10!).
-

Fig. 6.12. Planning the 'score' PROCedure.

number variable 'mark' will be incremented, and we can go back to the main program. More is needed if the answer does not match exactly. We need to print a message, and allow another go. When we do this, we must decrement 'try', otherwise it will be counted as another question, rather than as a free second shot. If the result of this next go is not correct, that's an end to the attempts. At this point, why not include some sound. We could have a different beep for a correct answer, first mistake and second mistake. Put it all in the plan!

Figure 6.13 shows the PROCedure 'score' that has been developed from this plan. It is placed in a REPEAT loop to allow for a second chance at each question. Lines 1220 to 1260 deal with a correct answer. If the answer is not correct, though, line 1270 checks to see if this is your second attempt, and if it is, lines 1280 to 1310 deal with it. If it's your first attempt at this question, then the PROCedure passes to line 1320. This goes on to deliver a message, then records your attempt in line 1340 and ensures that you get a free second shot at it. This is done by

```

1200 DEFine PROCedure score
1210 REPeat tests
1220 IF X$=A$ THEN
1230 mark=mark+1
1240 AT 2,6:PRINT"Correct- your score
    is now ";mark
1245 PRINT" - in ";try;" attempts."
1250 BEEP 20000,11
1260 EXIT tests
1270 ELSE IF Attempt=1 THEN
1280 Attempt=0
1290 AT 2,6:PRINT"No luck- try the ne
xt one"
1292 BEEP 15000,100
1300 PAUSE 100
1310 EXIT tests
1320 ELSE IF Attempt=0 THEN
1330 AT 2,6:PRINT"Not correct- but it
    might just be "\"your spelling. You
get another "\"chance, free."
1335 BEEP 10000,80
1340 Attempt=1:try=try+1
1350 PAUSE 100
1360 game
1370 END IF
1380 END REPeat tests
1390 END DEFine

```

Fig. 6.13. The 'score' PROCedure written.

repeating the 'game' call in line 1360, and when you have answered, the PROCedure loops back (because of the END REPEAT tests) to check this answer.

Now that we've got the bit between our teeth, we can polish off the rest of the PROCedures. Figure 6.14 shows the PROCedure 'setup' that deals with dimensioning and arrays. Line 1510 sets all the variables for the scoring system to zero. Line 1520 dimensions the array Q\$ that

```

1500 DEFine PROCedure setup
1510 try=0:mark=0:Attempt=0
1520 DIM Q$(10,20)
1530 RESTORE: FOR J=1 TO 10
1540 READ Q$(J):NEXT J
1550 END DEFine

```

Fig. 6.14. The 'setup' PROCedure for dimensioning and setting up the array.

will be used for the names of the animals and lines 1530 and 1540 then read the names from a data list into the array Q\$. We shall write the DATA lines later, as usual.

Figure 6.15 is the PROCedure for the instructions, and Fig. 6.16 is the title PROCedure. The title lines include a pause, and have been written with the CSIZE 3,1 type of display. This gives large letters with no hassle, and is excellent, unless you use long words that fall off the end of the screen! The instructions have been written as single lines

```

2000 DEFine PROCedure instructions
2010 CLS:AT 13,1:PRINT"INSTRUCTIONS"
2020 AT 1,3:PRINT"The computer will s
supply you with"
2025 PRINT"the name of an animal. You
are asked"
2030 PRINT"to type the name of its yo
ung. Make"
2035 PRINT"sure that your spelling is
correct,"
2040 PRINT"and that you start with a
capital"
2045 PRINT"letter. The computer will
keep the"
2050 PRINT"score for you. You get two
tries at"
2055 PRINT"each name."
2060 PRINT:PRINT" PRESS THE SPACEB
AR TO START"
2065 REPeat loop
2070 IF KEYROW(1)=64 THEN
2075 EXIT loop
2076 END IF
2080 END REPeat loop
2090 END DEFine

```

Fig. 6.15. The instructions PROCedure – always leave this until you have almost finished. The lines have been kept short to avoid corrupting memory.

```

3000 DEFine PROCedure title
3010 CLS:CSIZE 3,1
3020 AT 7,1:PRINT"Young Animals"
3030 PAUSE 300
3040 CSIZE 2,0
3050 END DEFine

```

Fig. 6.16. The title, which uses large letters.

because, as I explained earlier, a long instruction line seemed to corrupt the memory of my QL, so that it would not correctly record or replay the program. Finally, Fig. 6.17 shows the DATA lines.

```

6000 DATA "Dog", "Cat", "Cow", "Horse", "
Hen", "Fox", "Kangaroo", "Goose", "Lion",
"Pig"
6001 DATA 5,80,117,112,112,121
6002 DATA 6,75,105,116,116,101,110
6003 DATA 4,67,97,108,102
6004 DATA 4,70,111,97,108
6005 DATA 7,67,104,105,99,107,101,110
6006 DATA 3,67,117,98
6007 DATA 4,74,111,101,121
6008 DATA 7,71,111,115,108,105,110,10
3
6009 DATA 3,67,117,98
6010 DATA 6,80,105,103,108,101,116

```

Fig. 6.17. The DATA lines that are needed, for questions and answers.

Now we can put it all together, and try it out. Because it's been designed in sections like this, it's easy for you to modify it. You can use different DATA, for example. You can use a lot more data — but remember to change the DIM in line 1520. You can make it a question-and-answer game on something entirely different, just by changing the data and the instructions. You can create much more interesting sound effects, or add more interesting graphics. One major fault of the program is that once an item has been used, it can be picked again, because that's the sort of thing that RND can cause. You can get round this by swapping the item that has been picked with the last item (unless it was the last item), and then cutting down the number that you can pick from. For example, if you picked number 5, then swap numbers 5 and 10, then pick from 9. This means that the RND(1,10) step will become RND(1,D), where D starts at 10, and is reduced by 1 each time a question has been answered correctly.

There's a lot, in fact, that you can do to make this program into something a lot more interesting. The reason that I have used it as an example is to show what you can design for yourself at this stage. Take this as a sort of BASIC 'construction set' to rebuild any way you like. It will give you some idea of the sense of achievement that you can get from mastering your QL. As your experience grows, you will then be able to design programs that are very much longer and more elaborate

than this one by a long way. By that time, you'll know much more about the QL and about programming than I could show you in the space of one book.

Chapter Seven

Channels, Microdrives and Data Filing

The QL, unlike most other microcomputers, comes with a storage system for programs and data built in. Whereas most other computers use ordinary cassettes or floppy disks for such storage, the QL uses the Microdrive, a variation on the type of endless tape system that former owners of a TRS-80 or Video Genie may remember. Since the use of such a system will probably be quite new to owners of a QL, even if they have used a computer previously, this chapter is devoted to the use of the Microdrives for storing data. Chapter 1 has already dealt with the use of the Microdrives in connection with storing and loading programs.

The most puzzling new names that cause difficulty to the new QL owner are 'channels and devices'. Once you grasp what is meant by these words, and how they are applied, you will find Microdrive operation much more interesting, and you will also be able to do much more with your QL. Let's start, then, by explaining these words.

A *device* is something that puts out or receives data. Your keyboard is a 'device', because each time you touch a key, a set of electrical signals is sent to the computer. The screen is another device, because every time the computer sends a set of electrical signals to the TV set, you will see something on the screen. The keyboard is a transmitting device, because it sends signals. The screen (really the TV) is a receiving device, because it accepts signals. The QL screen, in fact, behaves as if it were three devices, because it can be used in three separate portions. Later, in Chapter 8, we'll see how you can divide up the screen space as you want.

Some devices can perform both operations. A console is the combination of keyboard and screen, which both sends and receives data. The Microdrive is also a device which can be used in both directions. The network connections allow you to connect more than one QL together, so that one QL can be either a transmitting or a receiving device for the other. When we use commands that refer to

these devices, we use abbreviations such as SCR for screen and MDV for Microdrive.

We have talked of electrical signals passing from one device to another, and this is what actually happens. It's a lot more useful to think of what these signals represent. Each set of signals represents a unit of data called a 'byte', and data is the stuff that computers are designed to deal with. Data may be numbers or names, it's anything that the computer has to work with. If you have a program that arranges the names of your friends in order of birthdays, then that program needs data. The data in this example is the set of names and birth dates. If you have a program that shows cookery recipes and shopping lists, then the data is the instructions, the names of the foodstuffs, and the quantities. Every computer that is designed to be used for anything more than the simplest games must be able to save and load data of this type separately from the program that generates it or uses it.

There's a lot to be gained from this approach. The memory of the QL is used for quite a lot of purposes over which you have no control. A very long program which gathered data and then made use of it might not fit into your computer. It's a lot more sensible to have a short program which gathers the data, using INPUT lines, and which records the data as it is gathered. The data is then safe if anything should happen that scrambles the memory of the QL. Another program can then make the use of this data. By keeping the two programs and the data separate, you can deal with a lot more information than would be possible if you had to have the whole lot in the memory at one time.

What has all this to do with channels and devices? Well, *devices* are the parts of the computer which give out or receive data, and *channels* are the paths which carry the data. Just think of what happens when you use your QL. When you press a key, something appears on the screen. The keyboard is one device, the screen is another, and there is a channel which links them. This is just a fancy way of saying that there is a path for data signals from the keyboard to the screen. The important point, however, is that these paths or channels are numbered, and we can control them. Controlling them means that we can change the paths, breaking some and making others, as we please. It wouldn't be sensible to break some of the paths, of course, except for special purposes. You normally want to see on the screen the words that you type on the keyboard. If you were typing a special password, however, and you didn't want anyone who was watching the screen to see it, it would make sense to break the channel that connects the keyboard to the screen. As it happens, some of these channels are not quite so easy

to break, simply because it might be just too easy to disable the computer in that way!

As you might expect by now, there are some channels which are connected to devices from the moment you switch on. It's obvious, for example, that there's a connection between the keyboard and the screen. If you think about this a bit more, you'll realise that more than one channel must be used. When you press a key, the result shows on the lower part of the screen, but a PRINT instruction or command causes words to appear on the *upper* part. The way that the QL is designed, sixteen channels can be used. Of these, four have fixed jobs to do, and you should normally try to avoid changing these tasks. The devices are identified by number, using the hashmark (#) before each number. The use of the hashmark is a reminder of the US origin of our computing languages; # is used in the USA in the same way as we use the abbreviation 'No.' for 'number'. When you read #3 in an American catalogue, then, it means what No. 3 would mean in a UK catalogue.

As so often happens in computing, the sixteen channels are not numbered 1-16, but 0-15. Of these channels, 0-2 are set aside for the QL's own use. Channels #0 to #2 control the three parts of the screen, and you should not interfere with them. If you type, for example:

PRINT#0,"test"

then when you press ENTER, you will see the word 'test' appear on the bottom line of the screen, not on the top where a normal PRINT statement places it. If you use PRINT#1,"test", then the word is printed in the usual position that you get when you use PRINT by itself. If you use PRINT#2,"test", then the word appears as if it were part of a listing, on the listing portion of the screen. If you are using a TV receiver rather than a monitor, then this means that the word appears in red letters on a white background rather than the other way round.

These channels #0 to #2 are used automatically when the QL performs any output to the screen. When you use PRINT, for example, you do not have to specify that the data goes to the screen using channel #1, the computer does this automatically. This saves you from a bit of extra typing at least. If you want to make text appear on the other portions of the screen, you now know how! Channels #3 to #15, however, are not normally used. Channel #3 appears to be commandeered at times when the Microdrive is operating, but it is restored to you afterwards. These channels exist for you to make your own uses of, and when you do so, you will be able to make data flow the way that you want. In order to do this you have to know two things. One is how to select a channel, the other is how to link it to a device.

The way that you select a channel is by using the OPEN command. For some purposes, such as selecting the serial port for a printer, this is followed by a hashmark, and then the number of the channel. If you type OPEN#3,SER1 for example, you will have prepared channel No. 3 to send data to a printer (see Appendix B) along serial port number one. Rather more than this is needed if you want to store data on a Microdrive cartridge, or load data from one, and that's what we'll look at next.

Data filing techniques

What is a file?

The word 'file' occurs many times in the course of this book. A file can mean any collection of characters which belong together. The characters of a BASIC program constitute a file, for example, because the program will not run if characters are missing. A set of names and addresses in ASCII code is a file, because they form one group of information, such as our friends, or our suppliers, or debtors. A set of bytes of machine code is a file, and a collection of the numbers that are used by a financial program is a file. In this Chapter, though, I'll take 'file' to mean a collection of information which we can record on a Microdrive cartridge. I'll also take it to mean a collection of data that is separate from a program. For example, if you have a program that deals with your household accounts, you would need a file of items and money amounts. This file is the result of the action of the program, and it preserves these amounts for the next time that you use the program. Taking another example, suppose that you devised a program which was intended to keep a note of your collection of vintage 78 rpm recordings. The program would require you to enter lots of information about these recordings. This information is a file, and at some stage in the program, you would have to record this file. Why? Because if a program like this is going to be really useful, there will not be enough space even in the memory of your QL computer to hold all of the information at one time. This is the topic that we're dealing with in this chapter, recording the information that a program uses. The shorter word is *filing* the information.

You can't discuss filing without coming across some words which are always used in connection with filing. The most important of these words are 'record' and 'field' (Fig. 7.1). A record is a set of facts about one item in the file. For example, if you have a file about LNER steam locomotives, one of your records might be used for each locomotive

FRIENDS FILE	
RECORD 1	
FIELD 1	Name 1
FIELD 2	Address 1
FIELD 3	Phone No. 1
FIELD 4	Birthday 1
RECORD 2	
FIELD 1	Name 2
FIELD 2	Address 2
FIELD 3	Phone No. 2
FIELD 4	Birthday 2
RECORD 3	
etc.	

Fig. 7.1. A data file, showing what is meant by records and fields.

type. Within that record, you might have designer's name, firebox area, working steam pressure, tractive force ... and anything else that's relevant. Each of these items is a *field*, an item of the group that makes up a *record*. Your record might, for example, be the Scott class 4-4-0 locomotives. Every different bit of information about the Scott class is a field, the whole set of fields is a record, and the Scott class is just one record in a file that will include the Gresley Pacifics, the 4-6-0 general purpose locos, and so on. Take another example, the file 'British Motorbikes'. In this file, BSA is one record, AJS is another, Norton is another. In each record, you will have fields. These might be capacity, number of cylinders, bore and stroke, suspension, top speed, acceleration ... and whatever else you want to take note of. Filing is fun – if you like to arrange things in the right order.

Microdrive filing

In this book, because we are dealing with the QL Microdrive systems, we'll ignore filing methods that are based on DATA lines in a BASIC program. This is because many buyers of the QL machine will be starting from scratch with a Microdrive system. Others may only have used cassettes for filing. If it's all familiar to you, please bear with me until I come to something that you haven't met before. To start with, there are two types of files, only one of which we can use with a Microdrive system. These are serial files and random access files. The difference is a simple, but important one. A serial (or sequential) file

records all the information in order on a Microdrive, just as it would be placed on a cassette. If you want to get at one item, you have to read all of the items into the computer, and then select. There is no simple way in which you can command the system to read just one record or one field. A random access file does what its name suggests – it allows you to get one selected record or field from the recorded data without reading every other one from the start of the file. The difference between serial filing on tape and random-access filing on disk is illustrated in Fig. 7.2. Random access filing is something which requires the use of a disk system, but we can design programs which achieve something like the same effect by using serial files on the Microdrive. We'll start, then, by looking at serial files, which are also the type of files that we record on a cassette.

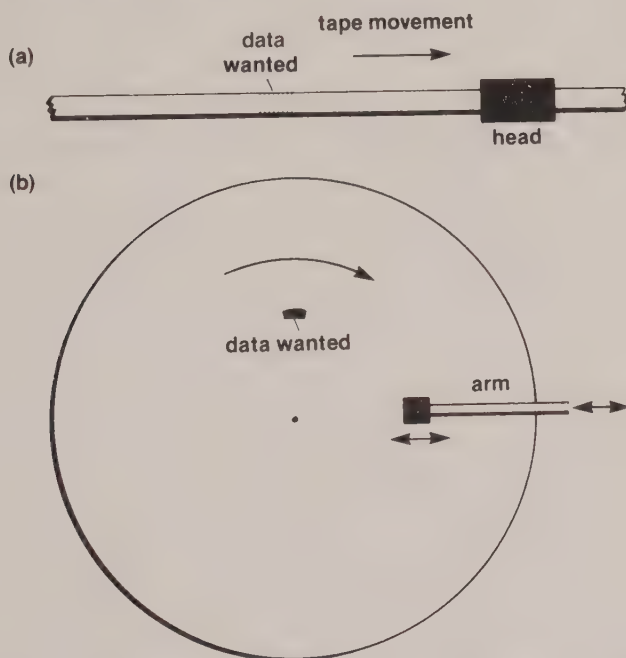


Fig. 7.2. Illustrating the difference between (a) serial files on tape and (b) random access files on disk. Any part of the disk can be reached by placing the head at the correct radius as the disk spins. To read a piece of tape, you have to pull at the preceding length of tape past the head.

Creating a file

When you are dealing with something new, it's always a good idea to start at the beginning and keep things simple at first. We'll start the idea

of filing with the way that we make connections to the Microdrive. Figure 7.3 shows a very simple example of how one item of data, the value of a variable called 'A', can be recorded on Microdrive number 1. Because the steps are so important, we'll look at each of them in close detail. Starting with line 10, then, we 'open a file'. This requires the

```
10 OPEN_NEW #7,MDV1_DATATEST
20 A=5
30 PRINT#7,A
40 CLOSE #7
```

Fig. 7.3. Recording the value of a single variable.

command word `OPEN_NEW`, and we have to specify a channel number (take your pick, 3 to 15), the number of the microdrive, and a name for this file of data. Line 10 uses `OPEN_NEW #7,MDV1_DATATEST` to open a new file on Microdrive number 1, using channel 7, with the name of `DATATEST`. If you use `OPEN_NEW` on a file which already exists on a cartridge, you will get an error message. This is to prevent you from 'wiping' a file by accidentally recording over it another one which has the same name. Similarly, if you use `OPEN_IN` in a program, using a cartridge which has no file recorded, you will also get an error message.

The next step is to assign a value to the variable, `A`, in line 20. Line 30 is the important one now. The instruction `PRINT#7,A` means 'send out the value of `A` over channel 7'. Channel 7, however, is connected to the Microdrive 1 because of the action of line 10, so that this line 30 will 'print' the value of `A` on to Microdrive No. 1. It will record the value, and make sure that it is filed under the name of 'DATATEST' so that it can be easily found again. As it happens, line 30 does *not* actually cause the Microdrive to carry out a recording. The Microdrive records groups of characters, and the operating system is arranged so that the computer will gather up data in the memory until it has enough. This part of memory is called a 'buffer', and when you open a channel, you also automatically allocate a buffer for that channel. When there is less than a full buffer load of characters to record, the data is recorded only when the program 'closes the device' as an indication that there is no more to come. This is done in line 40 by the `CLOSE#7` command.

Now what happens when this runs? As far as you are concerned, it's just that the Microdrive spins for rather a long time, then stops. The cursor reappears long before the Microdrive stops, however. This is because your data is being held in the 'buffer', buffer number 7 in this case since we used #7 in lines 10 and 30. Whenever the data has been

delivered to the buffer, the computer is ready for other orders, while the Microdrive gets on with its job independently – so the cursor reappears. Now we have to prove that this data was actually recorded, and show that we can recover it. Take a look now at the listing in Fig. 7.4. Line 10 in this listing uses OPEN_IN, not OPEN_NEW. We already have a file on our Microdrive cartridge, and we want to read it, not to create a new file. The device number this time is #5, just for a change, and we have to specify the MDV1 and the name of the file. Remember that the underscore sign (_) is *essential* when you are adding a filename to the Microdrive number.

```

5 CLS
10 OPEN_IN #5,MDV1_DATATEST
20 INPUT#5,A
30 PRINT A
40 CLOSE #5

```

Fig. 7.4. Reading the value back from the Microdrive tape.

Having ‘opened the file’, meaning that device #5 is now the one that will be used for reading this Microdrive, we can read the data. Line 20 does so, using INPUT#5,A. INPUT by itself always refers to the keyboard, but when we place the #5 after INPUT, it will cause the input to come from whatever is connected to Channel 5. We know what that is – it’s MDV1, reading the file called DATATEST. Line 30 prints what is read in, and line 40 closes the channel down again. It all looks reasonably simple and straightforward, but take a close look at these two little pieces of programming, because they contain a lot that you will need to get to grips with in the course of data filing.

Now try something more ambitious with the creation of a file of numbers. Figure 7.5 shows a program which generates a ‘file’ of numbers – the even numbers from 0 to 50, and then records these numbers on the Microdrive and also prints them on the screen. There are only six lines to this program, but three of them contain these important commands that you need to understand. We’ll start, reasonably enough, with line 10. This is one of these OPEN_NEW

```

10 OPEN_NEW #5,MDV1_EVENS
20 CLS:FOR N=0 TO 50 STEP 2
30 PRINT #5,N
35 PRINT N;" ";
40 NEXT N
50 CLOSE #5

```

Fig. 7.5. Recording a file of numbers.

commands which connects a device to a channel. The channel is #5, and we could have used any of the numbers from 3 to 15. The device to which we have connected #5 is the Microdrive No. 1, because we have used MDV1, and a filename of 'evens'. Notice that when you use the Microdrive as a device, you have to specify the drive number and the filename.

Line 10, then, connects #5 to a Microdrive file in Drive 1, with the filename of 'evens'. This will cause the Microdrive to spin so that a place is found on the tape for the data and the filename can be recorded at the start of this space. The next thing is to create a file, and in this case, it's being done by a loop which starts in line 20. This allocates variable N as each of the even numbers in turn, with the NEXT N in line 40. How do we place them into the channel? Line 30 does this, using PRINT #5,N. The PRINT #5 instruction will send the codes for the numbers along channel 5 – and that's the channel which is connected to the Microdrive file.

Now just sending data along the channel does *not*, as I explained before, cause it to be recorded on the Microdrive. The Microdrive is much faster than a cassette, but it's nothing like as fast as the computer. The FOR...NEXT loop in lines 20 to 40 could easily be completed before the motor of the Microdrive could be started! Each time line 30 runs, the value of N is stored temporarily in the 'buffer'. It's a good name, because its action is to connect the computer and the Microdrive so that they work smoothly together. In this simple example, all the numbers that are generated by the action of the loop are simply passed into the buffer. Nothing is recorded in this time; the Microdrive doesn't start turning. Line 50 is the one that actually causes the recording of data to take place. CLOSE #5 means 'close down channel 5', and when a channel is closed, part of the action is to empty any buffer which is part of that channel. Because the channel is connected to the microdrive, emptying the buffer is done by starting the drive, recording the filename, then all the data in the buffer, and then closing off the channel. That's it! If you now RUN this program, you'll hear the Microdrive find the first clear place of the tape, and then it will record the data. You can tell when it gets to this stage, because line 35 will print the numbers on the screen while the buffer is being filled by line 30.

You can think of the buffer as a shock absorber, a reservoir, a bank account ... anything that sits between an input and an output so that the inputs and outputs can be at different times. Your QL will create a buffer whenever a device is opened in line 10, and the size of the buffer is large enough for a lot of data. You can try it for yourself by altering the program so that it looks like Fig. 7.6. This time, the number of bytes of


```

10 OPEN NEW #6,MDV1_MOREVENS
20 CLS:FOR N= 0 TO 5000 STEP 2
30 PRINT #6,N
40 PRINT N
50 NEXT N
60 CLOSE #6

```

Fig. 7.6. A much longer file. The Microdrive runs continuously, emptying the buffer as the program fills it.

data needed will be several thousand bytes, and when you RUN the program, you will hear the Microdrive run continually until some time after all the numbers have appeared on the screen. What is happening is that the buffer is being filled because of the loop, and is being emptied by the Microdrive. Because the buffer memory is so large, it never fills. When the loop is completed, there may still be data in the buffer, and this is recorded when line 60 runs.

You can see from this description that line 60 is very important. Without it, small amounts of data are simply not recorded at all, and larger amounts of data are never completely recorded. There's something else as well. The CLOSE action, as well as recording everything that is left in the buffer, also puts a signal of its own on the tape. This is an 'end-of-file' signal, and it's used by the Microdrive to detect where the end of a file comes. If the Microdrive could not find this signal, it would not be able to detect where a file ended, and it might start reading another file. Leaving out the CLOSE command is called 'leaving your files open', and you wouldn't want to be seen with your files open, would you?

These short programs have put data into a file, but so far, you have had to trust me that there is actually something on the tape. Now we'll look at how we recover data from the tape in a moment, but this is a good chance to see how we can check data simply and easily. Type this direct command:

COPY MDV1_EVENS TO SCR

When you press ENTER, you will hear the Microdrive spin into action, and after a short time, you will see the even numbers appear on the screen. The screen will scroll, however, and you will see only the last of the numbers displayed by the time that the Microdrive stops spinning. You can also use this method to see what happens with a long file. Type:

COPY MDV1_MOREVENS TO SCR

and press ENTER. You will find that the Microdrive runs for a short time, delivering numbers to the buffer, and the buffer at the same time displays the numbers on the screen. It's the screen which is the hold-up this time, and you will find that it takes much longer to display the numbers than to read them from the Microdrive.

Take a good look at this bit of single-line magic. We have used COPY to link one device to another. The 'source' device is the Microdrive file on drive 1, the destination device is SCR, which means the screen. The effect of the command, then, is to put all of the data from the Microdrive file on to the screen so that you can read it. It's a very handy trick to remember, because it allows you to inspect what is in a file without needing to load a program that deals with the file. It allows you only one read each time the command is carried out, though, and you have to read the file in order. If you want to inspect part of the file, you will have to stop the reading, using CTRL and SPACE, and then start all over again when you want to see more.

The use of COPY is not exactly satisfactory if you want to see more of your data, and Fig. 7.7 shows a different method. In this example

```

10 OPEN_IN #7,MDV1_MOREEVENS
20 FOR N=0 TO 5000 STEP 2
30 INPUT #7,N
40 PRINT N
50 IF N/40=INT(N/40) THEN
60 FOR J=1 TO 1000:NEXT J
70 CLS
80 END IF
90 NEXT N

```

Fig. 7.7. Reading stored data one screenful at a time.

also, you'll see another example of the buffer in action. As usual, the program starts in line 10 with opening the file, using OPEN_IN, with the correct file name of MDV1_MOREEVENS. Instead of using COPY, however, we read the file with INPUT, using this time channel #7. Now if we simply used a loop to read and display the data, the buffer would be filled from the Microdrive and emptied by the screen display, and you would hear the Microdrive spin briefly, then see the numbers appear just as before. In this case, though, we have interrupted the process. Lines 50 to 80 perform the interruption. In line 50 the condition is IF N/40=INT(N/40). This means the condition when N divides evenly by 40. If N/40 has a remainder, then it can't be equal to INT(N/40), which is the whole number part of N/40 only. Each time N has a value that divides evenly by 40, then, line 60 causes a

delay, and at the end of the delay, the screen is cleared before the main loop continues. Now when you run this one, you will hear the Microdrive stop and start at intervals. While the Microdrive runs, the buffer is being filled, but when line 60 runs, the INPUT #7 step is not being executed, so the Microdrive simply stops and waits. You will hear this stop and start action going on all the time that data is being read back from the MOREVENS file. If you interrupt this action by pressing CTRL and SPACE together, you will find this time that you can resume normal service by typing CONTINUE (then press ENTER).

More serial filing

Suppose that what we want to record is not a set of numbers that has been generated by a program, but a set of names that you have typed. As far as the Microdrive is concerned, this is just another set of data, and it's dealt with in exactly the same way. Each time you press ENTER at an INPUT step, the data is stored in a buffer, and it stays there until the buffer is full, or until the entry is complete and the file is closed. Once again, you can see the importance of using a buffer – you wouldn't expect the Microdrive to record each letter as you typed it, would you?

Figure 7.8 shows a short program of this type. Normally, if you were gathering information like this, you would store the names as an array. This, as you probably know, introduces complications like having to dimension the array for the maximum length of entry (the longest

```

10 OPEN_NEW #6,MDV1_NAMES
20 CLS:AT 16,2:PRINT"Names."
30 PRINT"This program stores names o
n the\"Microdrive. Please type each
name\"when requested. Type X to end
the\"entry of names."
40 REPEAT NAMEINPUT
50 INPUT"Name is _ ";nm$
60 IF nm$="X" OR nm$="x"THEN
70 EXIT NAMEINPUT
80 END IF
90 PRINT#6,nm$
100 END REPEAT NAMEINPUT
110 CLOSE#6

```

Fig. 7.8. Storing a set of names on the Microdrive.

name) and the maximum number of names. Unless you might want to look at a previous entry at some time when you were entering names, however, you don't need to use an array for recording a set of names. It can be a different matter when you play back, but that's something that we'll look at shortly.

Taking the program in more detail now, line 10 does the device connection act that should be reasonably familiar to you by now. The channel in the example is #6, and this is opened to the Microdrive device for the file "NAMES". When the program runs, this line will cause the Microdrive to look for a blank piece of tape, record the filename of 'NAMES' at the start of this piece of tape and prepare to record the data. Lines 20 to 30 are concerned with brief instructions. The main REPEAT loop then starts in line 40, and the idea is to input and record each name until an 'X' or 'x' is typed. If you wanted to expand this into a more realistic name file, you would probably want more detailed instructions. You can, incidentally, enter your names in any form you like, even in forms which are rejected by some computers, such as ATKINS, TOMMY.

Line 60 tests for the entry of 'x', which is the signal that no more names are to be entered. If 'x' has not been entered, then line 90 places the name nm\$ into the buffer. If you type names fast and continually, you will find that the Microdrive spins continually, keeping up with the buffer. If you pause, then the Microdrive will stop, and you probably won't hear it start again until line 110 is executed. Once you have data recorded on a Microdrive, you will want to make a backup copy. This is done by placing your data cartridge in MDV1, a formatted spare cartridge in MDV2, and carrying out:

COPY MDV1_NAMES TO MDV2_

More on reading

By this time, you have a few files of both numbers and names stored on your Microdrive, and it's time to pay rather more attention to the methods that are used for reading files and using the information from them. Suppose, for example, that the numbers which we recorded in the file 'EVENS' had been placed on the file by an accounts program. They might, for example, be the daily takings of a small shop. One thing that we might want to do with the numbers, then, would be to read them from the file and add them, showing only the total. This is a conventional and straightforward piece of programming, and one for

which you will probably find a large number of uses. Figure 7.9 shows what is needed. It starts in line 10 by clearing the screen and then line 20 opens channel #7 to MDV1_EVENTS. Now when we recorded the file "EVENTS", we selected all the even numbers from 0 to 50, which is a total of 26 numbers. To read the same set back, then, line 40 uses a FOR...NEXT loop with 26 passes. The number 'Total' has been set to

```

10 CLS
20 OPEN_IN #7,MDV1_EVENTS
30 Total=0
40 FOR N=1 TO 26
50 INPUT#7,J
60 Total=Total+J
70 NEXT N
80 PRINT"Total is ";Total
90 CLOSE #7

```

Fig. 7.9. Reading and totalling a set of numbers.

zero in line 30. Line 50 then inputs each number from the file, giving it the variable name of 'J', and line 60 adds this number to the total. At the end of the loop, line 80 prints the value of the total and the file is closed in line 90.

Suppose that you didn't know how many numbers were recorded? This makes the use of a FOR...NEXT loop impossible, because you wouldn't know what number of passes to use. As it happens, we can cope with this in a file of this type so long as you know what the last item is. Since we know that the last item in this file is 50, we can read each number until we get to 50, and then close the file. The program of Fig. 7.10 shows this in action. Lines 70 to 90 are the important ones to watch here. So long as the number that has been read is less than 50, the program loops back to line 50 to read and add another number. When the last item of '50' is read, however, line 80 will be carried out, and then line 110.

What happens if you forget to test for the last item? Try it for yourself, by making line 70 read:

```
IF J=70 THEN
```

You will find when you RUN this that the Microdrive operates as usual, but you will get the message:

At line 50 end of file


```

10 CLS
20 OPEN_IN #8,MDV1_EVENS
30 Total=0
40 REPEAT data_in
50 INPUT #8,J
60 Total=Total+J
70 IF J=50 THEN
80 EXIT data_in
90 END IF
100 END REPEAT data_in
110 PRINT"Total is ";Total

```

Fig. 7.10. Detecting the last item in a file so as to stop the reading loop.

on the bottom of the screen, and the top part of the screen will remain blank. The information has been read and used, as you can see if you now type GO TO 110. The error message is a reminder that the whole of the file has been read, and so the command in line 50 cannot be carried out again. It's not particularly convenient to have your program stop with an error message, and it would be useful if there were a BASIC instruction which allowed the program to go to line 110 (in this example) automatically when the end of the file was reached. A lot of disk systems, for example, allow you to test for the end-of-file code, and go to the next part of the program when this is found. On my QL, the line 70 IF EOF THEN was accepted, but had no effect. The alternative, 70 IF J=EOF caused the machine to lock up completely so that it had to be reset.

This means that you have to put a special 'end-of-file' character in for yourself. This will be in addition to the one which is put in automatically by the Microdrive, and which is used by the Microdrive. The point of having your own 'end-of-file' character is so that you can read numbers or letters from a file until this character is found, and then close the file so that you don't get this 'End of file' error message. In the example, we knew that the last item in the list of numbers was 50, and we used this as our end-of-file. For your own programs which use numbers, you could choose your own number. If you were using a file of numbers, and you know that there would never be a '999' entry, then you could make this the last number in any batch. The number 0 is another possibility, if you could be sure that no other zero was stored in the file.

Another method that is useful is to record CHR\$(0) at the end of each file. Now this has to be read as a string, not as a number, so you might think that you could not use it if you had recorded a set of

numbers. As it happens, you can read the numbers as if they *were* a string! Figure 7.11 shows a program which creates a set of numbers in a file that is called "ENDIT". These numbers are just random whole numbers which will be between 10 and 100. Each one is added to a file, and after all of the numbers have been recorded, line 50 places CHR\$(0) at the end of the file as a marker. Line 70 then stops the program so that you have time to think about the next stage. What you have on tape is a set of numbers – but they are recorded as ASCII codes,

```

10 CLS:OPEN_NEW #6,MDV1_ENDIT
20 FOR N=1 TO 10
30 J=RND(10,100)
40 PRINT #6,J:NEXT N
50 PRINT #6,CHR$(0)
60 CLOSE #6
70 STOP
100 CLS:OPEN_IN #7,MDV1_ENDIT
110 REPEAT getthem
120 INPUT #7,J$
130 IF J$=CHR$(0) THEN
140 EXIT getthem
150 END IF
160 PRINT J$
170 END REPEAT getthem
180 CLOSE #7

```

Fig. 7.11. Using CHR\$(0) as an 'end-of-file' marker.

not in the form that QL normally uses for number variables. This means that we can read these numbers back from the file *as if they were string characters* rather than numbers. If you use J in place of J\$ in this program, you will probably find, as I did, that the QL locks up and responds to any LIST or NEW command with the error message 'bad line' – and you lose your program as usual!

You can now read the file back by typing CONTINUE and pressing ENTER. The Microdrive reads each number as a string, J\$, and carries out the test in line 130 before printing it on the screen. When the CHR\$(0) is found, the program moves to line 180 and closes the file. Until this end-of-file character is found, the program loops back to line 120 to pick up another string. Though the numbers are being read back in string form, they can still be used as numbers and we could, if we liked, carry out arithmetic on them.

Naming the names

Now that we have replayed and used a number file, it's time to start looking at some replaying methods for the file of names that we created earlier. When this file was created, each name was recorded, but there was no 'end-of-file' marker. This makes the file less easy to read unless you know how many names were used. You can, of course, use COPY as before to display the list, and Fig. 7.12 shows another method. This uses INKEY\$ to read each character at a time from the file. Line 20

```

10 CLS:OPEN_IN #6,MDV1_names
15 REPEAT getname
20 PRINT INKEY$(#6);
30 END REPEAT getname

```

Fig. 7.12. Using INKEY\$ to read a file.

prints the character that is read, and the semicolon ensures that the characters are printed on the same line. We could, of course, assign the characters to a string name instead of just printing. When the file was placed on the tape, a carriage-return (code 13) character was placed automatically between each name, so that you will see each name of the file on a separate line. This method of reading, however, will end with an error message, 'At line 20 end of file', and will stop the program.

As before, the remedy is to record a definite number of names, using a FOR...NEXT loop, or use an 'end-of-file' of your own. Here again, the CHR\$(0) marker is a useful one. Figure 7.13 shows an improved names program which records CHR\$(0) at the end of the file. As you will see when you try it, it allows you to record the names as you like, and it then replays the file for you to inspect, using CHR\$(0) to stop the reading process. Lines 200 to 290 will replay the file, checking for the end-of-file marker in line 240. Until the marker is found, the program loops back to line 230 to input another name, but when the marker is found, the file is closed.

Now this is all very well if all you want to do is to place names on the screen, but INKEY\$ or MOVE can do that just as well. What we normally want to do is to place the names into an array, so that the computer can make use of the data. Using an array allows you to carry out tasks like placing the names into alphabetical order, for example. Not all uses call for an array, however. Suppose, for example, that you want to search the names for one beginning with the letter 'J'. You could do this by using the program which is shown in Fig. 7.14.

The first two lines follow familiar patterns. When the program is run,

```

10 OPEN_NEW #6,MDV1_NEWNAMES
20 CLS:AT 16,2:PRINT"Names."
30 PRINT\"This program stores names o
n the\" Microdrive. Please type each
name\"when requested. Type X to end
the\"entry of names.\"
40 REPEAT namesin
50 INPUT \"Name is_ \";N$
60 IF N$=\"X\" OR N$=\"x\" THEN
70 EXIT namesin
80 END IF
90 PRINT #6,N$
100 END REPEAT namesin
110 PRINT #6,CHR$(0)
120 CLOSE #6
150 STOP
200 CLS:AT 16,2:PRINT\"REPLAY\"
210 OPEN_IN #7,MDV1_NEWNAMES
220 REPEAT readthem
230 INPUT #7,J$
240 IF J$=CHR$(0) THEN
250 EXIT readthem
260 END IF
270 PRINT J$
280 END REPEAT readthem
290 CLOSE #7

```

Fig. 7.13. An improved names file, using CHR\$(0) as a marker.

```

10 CLS:OPEN_IN #9,MDV1_NEWNAMES
20 AT 14,2:PRINT\"NAMEFINDER\"
30 AT 2,4:INPUT \"Letter is_ ? \";Q$
40 Q$=Q$(1)
50 REPEAT search
60 INPUT #9,N$
70 IF N$=CHR$(0) THEN
80 PRINT \"End of file- name not found
\"
90 EXIT search
100 ELSE IF Q$=N$(1) THEN
110 PRINT \"Name is \";N$
120 EXIT search
130 END IF
140 END REPEAT search
150 CLOSE #9

```

Fig. 7.14. Finding names in a file.

line 10 will have the effect of filling the buffer, so that if you have only a comparatively small amount of data, all of it will be in the buffer. Line 30 asks you for a first letter of the name. If you give more than a first letter (don't forget capitals!), then line 40 takes care of this by selecting the first letter only. Line 50 starts a loop that takes a name from the buffer and checks first for the end-of-file character. If the whole file has not been read, line 100 compares the first letter that you have selected with the first letter of the name. If you have searched through the whole list without finding this letter, then line 70 will respond by ending the program when the end of file marker is found. If, on the other hand, the name that you want is found, then line 110 will print it, and the search ends also.

This works quite well with small amounts of data, but is not so useful when the data consists of more than one buffer-full of names. If you have to load from the Microdrive to find each name, then the process is slow. For very large amounts of data you may have no real option. Many programs of this type, however, can be dealt with more satisfactorily by using an array to hold all the data in the memory. The Microdrive is then being used as a store only, and all of the selection is being done in the memory of the computer. You might wonder if there is any advantage here as compared to having the data in DATA lines. There is, because the data can be created by one program, and used by several others. You can make the program that reads and uses the data a fairly short one, so that there is room for a lot of data in the large memory of the QL. The Microdrive, in other words, allows you to use short programs and lots of data.

Figure 7.15 shows how data from a program that has been put into the Microdrive can be read back into an array. We could, of course, have recorded an array in the first place using an instruction of the form:

```
PRINT #9,A$(J)
```

but in this book, I want to concentrate on the techniques that are peculiar to the Microdrive, rather than revising ideas that you should know about already. If we are to read items into an array, we find ourselves facing a problem. The problem, you see, is that we have to dimension the array so that it will hold all of the items. To do this we need to know how many items there will be, and how many characters are in the longest name. If we didn't count either of these at the time when we recorded, what can we do? As it happens, the answer is quite simple. We read all the names, count them, measure their length, and note both of these numbers as variable values. This can be done by a


```

10 CLS:OPEN _IN #9,MDV1_NEWNAMES
20 AT 14,2:PRINT"NAMEFINDER"
30 AT 2,4:PRINT"Type the first letter
  of the name\"that you want. Type 0
  to stop the\"action."
40 J=1:L=0:no=0:getdata:DIM N$(no,L)
50 REPEAT entry
60 INPUT #9,A$
70 IF A$=CHR$(0) THEN
80 EXIT entry
90 END IF
100 N$(J)=A$:J=J+1
110 END REPEAT entry
120 Z=J-1
130 REPEAT comparison
140 REPEAT testlet
150 INPUT "First letter_ ? ";Q$
160 Q$=Q$(1)
170 IF Q$="0"THEN
180 EXIT comparison
190 END IF
200 M=0
210 FOR X=1 TO Z
220 IF N$(X)(1)=Q$ THEN
230 PRINT N$(X)
240 M=1
250 NEXT X
260 ELSE NEXT X
270 END IF
280 IF M=0 THEN
290 PRINT"Cannot find the name. Pleas
  e try\"another one."
300 END IF
310 END REPEAT testlet
320 END REPEAT comparison
330 CLOSE #9:CLEAR
340 PRINT"END OF PROGRAM"
350 STOP
360 DEFine PROCedure getdata
370 REPEAT test
380 INPUT #9,A$
390 IF A$=CHR$(0) THEN
400 EXIT test
410 END IF
420 CH=LEN(A$)

```

```

430 IF CH>L THEN
440 L=CH
450 END IF
460 no=no+1
470 END REPEAT test
480 CLOSE #9
490 OPEN_IN #9,MDV1_NEWNAMES
500 END DEFine

```

Fig. 7.15. An improved name finder, which reads the name into an array. The array is dimensioned correctly by reading the names and checking number and maximum length.

PROCedure. Having done this, we use these variable values as dimensions, and we go through all of the reading action again, this time dimensioning the array correctly. The only snag about this system is that we are forced to run round the whole tape between finding the dimensions and loading the data into the array. This takes only a short time on the QL Microdrive, and the time is pretty much the same whether we have a lot of data or very little. It's a small price to pay for the sake of precise dimensioning. Precise dimensioning means that you will never get an error message because of an attempt to place too many items into an array, and you will not get items with letters missing because the length dimension was not large enough. It also means that you are using the memory of the QL in the most efficient way, and that can make the difference between being able to use as many items as you need and being restricted to a lot less.

Figure 7.15, then, is the listing for this program. The important new steps are in line 40 and in the procedure called 'getdata', which is defined starting in line 360. In line 40, the number variables L (length) and no (number of items) are set to starting values. The procedure 'getdata' will then assign correct values for these quantities, and they are put into the DIM statement at the end of line 40. The procedure consists of a loop in which all the items are read, one by one, until the CHR\$ 0 end-of-file marker is found. Lines 430 to 450 find the maximum length that is needed. This is done by finding the length of each item and comparing it with the greatest length, L, that has been found previously. If the length of the latest string is more than the length of any previous string, then L=CH makes this length the new maximum value. By the time that all of the values have been read, variable L will have a value which is the maximum number of characters in any string. The variable 'no' is used to count the number of strings that are read in. This will not include the marker. We have to

be rather more fussy in this program than we were previously, because the dimensioning is exact. Line 120, for example, has to make $Z = J - 1$, because the counter J has been incremented *before* the end-of-file character is found, so that the value of J includes the end of file character.

The most noticeable point, however, is that we have to open and close the file *twice*. The file has to be opened to count the items and measure their length. Since this counting action in the procedure takes us to the end of the file, we then have to close it (you can't go back after finding the end of a file) before we can use another INPUT from this file. If you omit the CLOSE and OPEN steps in lines 480 and 490, you will get an 'End of file' message in line 60. It's not exactly an obvious thing to think of when you are designing a program, so this example may be useful to you.

Back in the main program at line 40, following the 'getdata' step, the array that is going to be used can be dimensioned, since we have read the number of items and the length of the longest string that is stored. The next step is to read the names from the Microdrive into an array. This is dealt with by the loop called 'entry'. The steps are simple – a string is read in line 60, tested for the CHR\$(0) terminator, and if it is not CHR\$(0) it is added to the array by line 100. At the same time, the array counter J is incremented. At the end of the 'entry' loop, J has a value which is one greater than the real number of items in the array, so that line 120 is used to correct this.

Line 130 then starts the main loop that is used to work with the array in this example. This loop extends to line 330, but it can be terminated by entering '0' (zero, not letter O) as the letter that you are looking for. The main action is in the loop 'testlet'. This asks you for the first letter in line 150. As before, this is chopped to one letter just in case you entered more than one letter, and line 180 allows you to jump out of the 'comparison' loop if a zero has been entered. Note that this also causes you to jump out of the 'testlet' loop and there aren't many varieties of BASIC around that would allow you to do this. The penalty you pay is that the memory may be corrupted, and you may have to include a CLEAR at the end of the program to get things right again. This is a point that I'll deal with in more detail in a moment. In line 200, the 'flag variable', M , is assigned to zero. A flag variable is one that is used to signal some event, and I'm using this one to signal when a name is found in the list – you'll see why later. Line 210 then starts a FOR...NEXT loop which includes all the items of the array. Lines 220 to 240 ensure that if a matching letter is found, the name is printed and the variable ' M ' is set to 1. Line 250 is NEXT X because when a name

with a matching letter has been found, you still want to test the rest of the list in case there is another suitable name. If you omitted this, then you would always get just the *first* name on the list that matched, no more. Line 260 provides a NEXT X for all the names that don't match.

Now we make use of variable M. If M is still zero after all this, it must be because no name with the correct initial letter is on the list. Lines 280 to 300 detect this, and print a suitable message. This brings us to the end of the 'testlet' loop, and we will then want to try another letter. This is achieved by making the next line, line 320, the END REPEAT comparison line. That's it!

Making amendments

Let's be clear from the start that you cannot alter a file that is recorded on the Microdrive. What you can do, though, is to read in a file, make some alterations to it, and then re-record it. Since the QL has two built-in Microdrives, you can take advantage of this feature to record the new file *using the same name* on the second Microdrive. The important point here is that you can update a file without having to change its name. The Microdrive system is arranged so that you cannot 'wipe' a file by recording another file of the same name over it. This means, however, that you can't record an updated file of the same name on the same Microdrive. With two drives, however, this is simple, and Fig. 7.16 shows how.

There are dangers, however, which are not clearly stated in the manual. I found while testing this program that I was getting corrupted files, and the only way I could avoid this was by placing the CLEAR command at places in the program. This is doubtless a special feature of the QL Microdrives, but it means that you have to be very careful how you test your programs which make use of the Microdrives for filing. Never trust valuable data to a Microdrive filing program until you have tested it really thoroughly.

The program starts with a 'PLEASE WAIT' notice, because the first half-minute or so will be taken up with the Microdrives whirring (or in my case, clattering) round, with nothing to show for it on the screen. This is the time when the NEWNAMES file on MDV1 is being copied, as it is, on to the MDV2 file, under that same filename. The CLEAR in line 20 was found necessary to avoid the names being corrupted with 'x' signs and £ signs in the middle of each name. Line 30 opens MDV1 for reading and line 40 opens MDV2 for writing the new file. The 'copylist' loop then reads each string from MDV1, tests for CHR\$(0) and, if the


```

10 CLS:AT 13,6:PRINT"PLEASE WAIT."
20 CLEAR
30 OPEN_IN #5,MDV1_NEWNAMES
40 OPEN_NEW #6,MDV2_NEWNAMES
50 REPEAT copylist
60 INPUT #5,A$
70 IF A$=CHR$(0) THEN
80 EXIT copylist
90 END IF
100 PRINT #6,A$
110 END REPEAT copylist
120 CLOSE #5:CLEAR
130 CLS:AT 13,2:PRINT"Adding Items"
140 PRINT"Type the items that you want
to add.\"Type the letter X to stop
entry."
150 REPEAT newentry
160 INPUT "Name, please_ ? ";A$
170 IF A$="X" OR A$="x" THEN
180 EXIT newentry
190 END IF
200 PRINT #6,A$
210 END REPEAT newentry
220 PRINT #6,CHR$(0)
230 CLOSE #6
240 CLEAR
250 DELETE MDV1_NEWNAMES

```

Fig. 7.16. Updating a file, using both Microdrives. A new copy of the data is created on the second drive, and the first copy is deleted.

string is not a terminator, records the string to MDV2. When the terminator of CHR\$(0) is found, the loop ends, *but the terminator is not recorded*. The reading channel is cleared, and another CLEAR is used to prevent the next set of names from being corrupted.

Entry of new names starts with line 130 providing a title, while line 140 gives instructions. The 'newentry' loop then starts. The INPUT step gets each name from you and records it on MDV2. As usual, you won't hear any activity until the buffer is full. If you have omitted the CLEAR in line 120, you'll get corrupted names, and an error message telling you that there is something wrong with line 210! You can enter names until you type an 'X' (or 'x'), which stops the entry by jumping out of the loop to line 220. This records the end-of-file marker, CHR\$(0), and the line 230 closes the channel. There is a final CLEAR just for good luck in line

240, and then the last line deletes the file called NEWNAMES on MDV1. It's a good idea to omit this step until you are quite certain that the QL is not scrambling your names for you!

You have to be clear why this deletion is carried out. When you have updated a file, the cartridge with the updated file is now your main file, and you'll put it into the MDV1 slot. Next time you update, you will put the original cartridge in the MDV2 slot. If you didn't delete 'NEWNAMES' in line 250, you could not move the data from the cartridge in MDV1 into the cartridge which is put into the MDV2 slot, because you would be trying to write to a file that already existed. You might think that I'm rabbiting on a bit about this point. If I am, it's because it's important. From experience, I find that newcomers to the Microdrive make this mistake more often than any other. This is particularly true if you have used a disk drive with any other machine. Most disk systems will quite happily write a new file all over an old one, using the same name. This can cause loss of files if you are not careful, so the system that is used for the Microdrive is a valuable safeguard.

Chapter Eight

Windows and Other Effects

Working with a computer is never dull, but a lot can be done to make things more interesting. One of the special effects that is now available on modern computers is 'windowing', and in this chapter we'll take a look at this effect and a few others that produce screen displays that are rather more interesting than plain text or figures. Windowing looks like a good place to start, because unless you have used one of the few machines that allows this effect to be programmed easily, you probably haven't tried this for yourself.

A *window* simply means a section of screen which can be used independently of other sections and even independently of the screen as a whole. A window can have text printed on it, and can be scrolled or cleared irrespective of what is happening on other parts of the screen. The use of windows on the QL is complicated by the fact that there is more than one command to create a window, and also by the difficulty of removing a window that has been created by one of the commands. It's time, then, to look at how we can control this window business for ourselves. Figure 8.1 illustrates, for a start, what can be done. When this program runs, you will see a listing. This has been done because it makes it easier for you to see the shape of each window. After the listing, a window appears at the top of the screen. You'll see that this window extends for almost the whole width of the screen, and there is no border such as you normally see with the main screen. Following a pause another window appears near the bottom of the screen. This, like the first window, has a red background colour, and you'll see that words that are printed on this window adapt to the shape of the window, just as if this were the whole extent of the screen. You will see the text in this window scroll, and then the window is cleared. The next step is to make a window of the size of the whole screen. Words printed on this window can be lost at the sides, because a TV receiver does not display the whole of the picture. The border is therefore restored, and another bit of text printed to prove it.

```

10 LIST
20 WINDOW 471,25,20,20:CLS
30 PRINT"This is the top window"
40 rest
50 WINDOW 200,30,156,200:CLS
60 PRINT"This is a lower window"
70 rest
80 PRINT"It scrolls like a full screen"
90 rest
100 PRINT"- and it clears with CLS"
110 rest
120 CLS:rest
130 WINDOW 511,255,0,0:CLS
140 PRINT:PRINT:PRINT"This gives full
1 screen- no border"
150 rest
160 BORDER 20,0
170 PRINT:PRINT:PRINT"Border is now included"
200 DEFine PROCedure rest
210 LOCAL N
220 FOR N=1 TO 1500
230 NEXT N
240 END DEFine

```

Fig. 8.1. Illustrating windows on the QL screen.

Now that little lot should have given you some idea of what can be done with windows, and what you need to know now is how it is all done. As we start to analyse the program, though, we'll have to look at a number of ideas which are essential to this principle of windowing, and which will turn up again in later chapters. The first of these ideas is screen co-ordinates, which we meet for the first time on line 20.

A co-ordinate system is a way of specifying positions with numbers. Like the column and line numbers for AT, it works like the old-style Treasure Island maps. One of these might tell you to start at the old oak tree, then take twenty paces North and ten West. In the language of co-ordinates, the old oak tree is the 'origin', the point from which everything is measured. The paces are the units of distance, and the directions are always at right angles. For the purposes of using the QL screen, the origin for WINDOW and several similar commands is a point at the *top left-hand corner* of the screen. This point may, if you use a TV receiver, actually be just off the screen and invisible to you.

The size of the steps is such that, no matter what size your TV screen is, there will always be 512 steps across it, and 256 steps down. It's as if your screen were invisibly divided into a grid of 512 by 256 lines (Fig. 8.2). We refer to the distances across as the 'X-co-ordinates' and the distances down as the 'Y-co-ordinates'. All of these distances, remember, are measured from the top left-hand side. A point of the screen which had an X co-ordinate of 255 (half of 512, less 1) would be halfway along the screen. Similarly, a point which had a Y co-ordinate of 127 (half of 256, less 1) would be halfway down the screen. Why do we subtract 1? Well, the co-ordinate numbers are not 1 to 512 and 1 to 256, but 0 to 511 and 0 to 255. This is a scheme that is followed to a large extent in computing, and you just have to get used to it. One step is a small distance across the screen, almost unnoticeable on a TV receiver, and so it won't matter if you take the centre of the screen as having co-ordinate numbers of 256 and 128 instead of 255 and 127. In fact, because TV receivers differ in the way that they are adjusted, you might find that neither set of co-ordinates gives you a point that is exactly slap-bang in the middle of the screen!

Now for a closer look at line 20 of the program, Fig. 8.1. The `WINDOW` command-word is followed by four numbers, with commas separating them. The first pair of numbers decide the *size* of the window, following the order of X, then Y which we normally use. This command, then, asks for a window of 471 units wide and 25 units deep. In the Y direction 25 units is big enough to allow a couple of lines of printed text, with space to spare. The next two numbers show the 'origin' of the window. This means that the two numbers are the X and Y numbers for the point at the top left-hand corner of the window. We have used 20 and 20 here, so that this window will start close to the top left-hand corner of the screen. The `CLS` command now clears the *window*. Notice that it has no effect now on the rest of the screen, only on this window. Similarly, a `PRINT` command affects only this window now. Once you have specified a window by using the `WINDOW` command, your screen commands apply only to this window, and the rest of the screen is unaffected.

How can this be changed? One way is to specify another window. This is done in line 50. This time, the window is narrower, 200 units. Since the maximum X range is 512, 200 is about $\frac{2}{5}$ of the screen width. The Y dimension is only 25, and the start of the window is at $X=156$, $Y=200$. By choosing $X=156$, I shall make the window appear centred on a line. This is because $512-200=312$, and $312/2=156$. Using 156 as the origin therefore means that we have 156 spaces before the start of the window, 200 spaces of window, and another 156 spaces on the

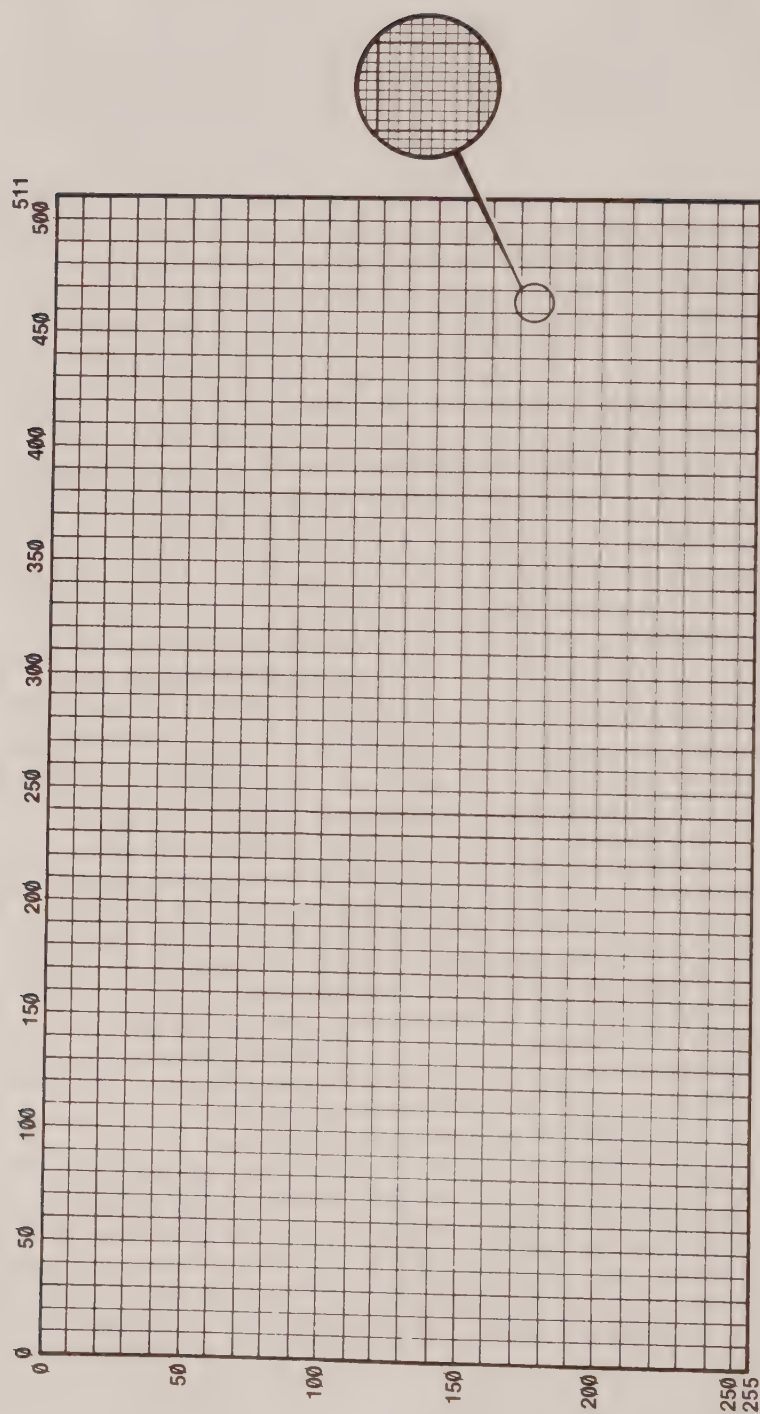


Fig. 8.2. The screen co-ordinate numbers for the instructions in this chapter. A different set of co-ordinates is used for graphics.

right-hand side. With $Y=200$, we get the origin of the window close to the bottom of the screen. The following lines then show how text appears in this window. Text which you may have typed in a form suitable for the large screen will be differently spaced in this window. It will scroll as needed, and when you perform a CLS, only this new window is cleared. The previous window at the top of the screen is unaffected by all this, as is the rest of the screen.

All this could result in a lot of parts of the screen being unaffected by anything you did, so there has to be a way out. Line 130 illustrates part of this. By using the largest possible co-ordinate numbers, and origin numbers of 0,0, we make the whole area of the screen into a window! The trouble now is that you may not be able to use all of it, especially with a TV receiver. You'll probably find that several letters of the phrase in line 140 are lost at the left-hand edge of the screen. This is because the protective border, which is normally placed round the screen when you switch on, is removed by this window. We can restore this border by the instruction in line 160. There are two numbers used in this BORDER command. The first number specifies the size of the border, 20 units. The second specifies the border colour, black. We'll come back to colours later, but if you omit this number, the border will be the same colour as the rest of the screen, and you won't know that you have a border until you see a complete window full of text. Lines 200 onwards, as you can see, define the procedure called 'rest' which provides a pause between actions. Yes, we could have used PAUSE, but there's nothing like a bit of variety.

Channelled windows

The WINDOW command is very useful, and we have still a few WINDOW tricks to use, but it can be awkward to lose command of a window when you define another window. Fortunately, there is another way round this. Another way of defining a window exists, but with the difference that the window is connected through one of the numbered 'channels' that we have met previously. In this way, we can always print to this window, and clear it, by specifying its channel number. This is true even if we have specified and used several other windows as well. The key to this type of window is the use of OPEN with a channel number, and the use of SCR to specify a screen.

Figure 8.3 shows this type of window in action. The whole screen is cleared, and then a window is connected to channel #4 by using the command:

```

10 CLS
20 OPEN#4,SCR_200X30A156X20
30 LIST #4
40 PAUSE 200
50 OPEN#5,scr_400x50a56x200
60 PAUSE 200
70 CLS #4
80 PAUSE 200
90 PRINT #4,"Text in top window"
100 PAUSE 200
110 CLS#5:PRINT #5,"Text in bottom wi
ndow"
120 PAUSE 200
130 PRINT#5,\\\\"
140 CLS#4:PRINT #4,"bottom window scr
olled!"
150 PAUSE 200
160 CLS
170 PRINT"Complete screen cleared"
180 CLOSE#4:CLOSE#5

```

Fig. 8.3. Connecting a window to a channel.

OPEN#4,SCR_200X30A156X20

Now what this means is that channel number 4 is being connected to a screen. The 200X30 part of the screen definition is the size of this screen (a window, in fact), in terms of X and Y co-ordinates. The use of 'X' as a separator is rather confusing here – you can't omit it. The 'A' which follows is another separator, because the next two numbers are, as you might expect by now, the co-ordinate numbers for the origin. Once again, the numbers have to be separated by an 'X'. Both sets of numbers are in the form of X co-ordinate followed by Y co-ordinate. The effect of this command, then, is to allow you to use commands like CLS #4 and PRINT #4 to clear this window or print to it. This action is selective. Unless you close the channel or allocate it to something else, it will be controlled by using channel #4. Another big difference is that the whole screen can still be cleared and printed on. Clearing the whole screen will also clear this window, and printing on the whole screen will print over the window.

In the example, then, the first window is created in line 20, and to prove that it works, the program is listed to this window. This is done by using LIST #4 – an ordinary LIST would have used the main screen. You can prevent the main screen from affecting your windows by using

the channel numbers for the main screen rather than the #4 and #5 that we have used here. The trouble with that is that it's not so easy to get back to normal when you want to. Getting back to the program, after the listing, there is a pause caused, not by a procedure this time but by a PAUSE. After the pause, another window is opened on channel #5. This window is near the bottom of the screen. The rest of the program then illustrates how the windows can be cleared and printed on separately. Note how the bottom window is scrolled in line 130 by sending a set of 'new-line' characters to the channel number. Notice too that the CLS in line 160 affects the whole screen, and clears the windows as well.

The secret scrolls

So far, you will have seen the screen scrolling as a program lists, or when you keep printing text. You can, however, control the scrolling of the screen by means of the SCROLL command. This is particularly useful to obtain effects that would otherwise be very hard to obtain, as Fig. 8.4 shows. In this example, you'll see that the word 'TEST', which fills the width of the window, is scrolled slowly. The first scroll is down

```

10 CLS
20 WINDOW 50,255,230,0
30 PRINT"TEST"
40 FOR N=1 TO 240
50 SCROLL 1
60 NEXT N
70 FOR N=240 TO 1 STEP -1
80 SCROLL -1
90 NEXT N

```

Fig. 8.4. Illustrating the SCROLL command.

the screen, and then the word is scrolled up the screen. This is done by using a number following the SCROLL command. The number that follows SCROLL indicated how many steps the screen will be scrolled. As before, we have 512 steps in the X-direction and 256 in the Y-direction. By using a FOR...NEXT loop which has 240 steps, and with SCROLL 1 in the loop, we move the text only by a very small amount on each scroll, $\frac{1}{256}$ of the screen height in fact. This makes the movement appear to be quite smooth. The use of SCROLL -1 makes the scroll reverse, so that the word appears to move up the screen. You

could slow down the SCROLL action by placing an additional PAUSE step in the loop if you liked, but the movement would not appear quite so smooth, although it still looks good.

We can move on from here to more scroll secrets. A second number can be used following SCROLL 1 (or whatever number of steps is specified), which controls the type of scroll. The type of scroll means that you might not want all of a piece of text to be scrolled. Take a look at the listing in Fig. 8.5 now. This starts pretty well as the previous listing started, with a clear screen and a window definition. Line 30 then places text into the window, using a procedure. The loop that starts in line 40 then causes a scroll, but the command is SCROLL 1,1. The extra '1' means that the scrolling will exclude any lines under the cursor.

```

10 CLS
20 WINDOW 100,255,206,0
30 textput
40 FOR N=1 TO 150
50 SCROLL 1,1
60 PAUSE 2
70 NEXT N
80 FOR N=1 TO 140
90 SCROLL -1
100 NEXT N
110 CLS
120 CURSOR 0,20
130 textput
140 CURSOR 0,20
150 FOR N=1 TO 250
160 SCROLL 1,2
170 NEXT N
180 STOP
190 DEFine PROCedure textput
200 PRINT"This\"is\"a\"test\"of\"
    \"the\"scroll\"action\"of\"the\"Q
    L\"
210 PRINT\"which\"will\"demonstrate\"
    \"what\"is\"possible\"
220 END DEFine

```

Fig. 8.5. Selective scrolling, using a second number following SCROLL.

Now the printing action has left the cursor on the last word, so each line above this starts to scroll. I have slowed the action down with a PAUSE in the loop. The scroll stops at a point which leaves the text still looking sensible, and then the whole phrase is scrolled up again by using

another SCROLL loop, this time without any additional number. So far, so good.

The next bit of magic, however, uses SCROLL 1,2. This will scroll all of the text *below* the cursor. This wouldn't produce anything interesting to see if the cursor were on the bottom line of text, so the cursor is shifted in line 120. The positioning of the cursor is carried out by using CURSOR 0,20. Now these numbers are the usual X and Y step numbers, but you have to note how they are used. When you use CURSOR X,Y, the numbers are relative to the origin of the window that you are using. In other words, CURSOR 0,0 will mean the top left-hand corner of the window that you are using, *not* the top left-hand corner of the whole screen. By using CURSOR 0,20, then, we have set the position of the start of the text on the left-hand side of the window and 20 steps down. The procedure then prints the text starting at this point. Line 140 then returns the cursor to the top line, so that you can see how the SCROLL action scrolls the lines under this one. We need not place the cursor at the top line, of course. You have to experiment a bit with cursor placement, however, because it's possible to place it halfway down a word, giving some very odd effects. Try making line 140 read CURSOR 0,76, for example!

There's another command, PAN, which can give you the effect of scrolling left or right. The use of PAN is so similar to the use of SCROLL that I won't give an example here – the manual is quite helpful with this one as well.

Some prettier printing

The best place to start on our next bit of exploration of special effects is with the PRINT modifiers. As the name suggests, there cause changes on the appearance of anything that is printed on the screen. This action is the same no matter what we print – letters, digits, or graphics shapes. Take a look for starters at the program in Fig. 8.6 which illustrates one of these 'effect' instructions, FLASH. FLASH is turned on when it is followed by a 1, and off when it is followed by a 0. Its effect will be seen on anything that you print between the FLASH 1 and FLASH 0. The flashing will continue until the line scrolls off the screen, or the screen is cleared, or the line is replaced by another one.

FLASH is a useful effect if you particularly want to bring the user's attention to something on the screen. A set of effects which are quite different are provided by the OVER command. The OVER command can be followed by one of three digits, 0, 1, or -1, and Fig. 8.7 illustrates


```

10 CLS
20 PRINT "This is an ordinary phrase"
30 PAUSE 150
40 FLASH 1
50 PRINT "This one is flashing"
60 PAUSE 100
70 FLASH 0
80 PRINT "This one is not"

```

Fig. 8.6. How to program flashing letters.

```

10 CLS
20 AT 2,4:PRINT "T I   S T E L O E
O M N "
30 PAUSE 200
40 OVER 1
50 AT 2,4:PRINT " H S I   H O   V R C
M A D "

```

Fig. 8.7. The OVER command.

the use of OVER 1. When OVER 1 has been used, any printing on the screen can be overprinted in the same colour of print. In other words, one letter can be placed on top of another, rather than replacing it, as is more usual. You don't often want to make the peculiar shapes that this can provide, but for some purposes it can be useful. Figure 8.7 illustrates an eye-catching way of printing a title, so that half of the letters are printed, followed after a pause by the other half! This can be useful for display work. Another use for OVER 1 is in creating accented letters for foreign correspondence (if you write to the Spanish QL club, for example). Figure 8.8 shows this type of use, with the OVER 1 allowing you to place the accents over the letters 'e' and 'o'. Another useful application is the printing of the 'plus-or-minus' sign that so often occurs in manufacturing. This is illustrated in lines 70 and

```

10 CLS
20 AT 4,4:PRINT "Lycee Superieur"
30 AT 4,6:PRINT "El lenguaje de comput
acion"
40 OVER 1
50 AT 7,4:PRINT " "
60 AT 26,6:PRINT " "
70 AT 10,8:PRINT "+ 2% Tolerance"
80 AT 10,8:PRINT " - "
90 OVER 0

```

Fig. 8.8. Using OVER to form accents and other marks.

80. The underline sign has to be used rather than the genuine minus sign, because the minus sign is not in the correct position. The effect is quite good, though, and it saves having to adjust the cursor position with the `CURSOR` command.

Figure 8.9 shows the curious differences between `OVER 1` and `OVER -1`. `A$` is a set of letters, and `B$` is a set of letter 'I' which will be printed over. On the first run, `OVER 1` is used, and on the second,

```

10 CLS:OVER 0
20 A$="00000VVVVVXXXXX"
30 B$="IIIIIIIIIIIIIIII"
40 AT 2,4:PRINT A$
50 PAUSE 150
60 OVER 1
70 AT 2,4:PRINT B$
80 PRINT"OVER 1 USED"
90 PAUSE 100
100 OVER 0
110 AT 2,8:PRINT A$
120 PAUSE 150
130 OVER -1
140 AT 2,8:PRINT B$
150 PRINT"OVER -1 USED"
160 OVER 0

```

Fig. 8.9. The differences between `OVER 1` and `OVER -1`.

`OVER -1` is used. You can see that the letters that are printed by `OVER -1` are of a different colour, and that the effect of the overprinting is to produce rather different looking shapes, on the TV screen at least. Oh yes, I nearly forgot `OVER 0`, which restores everything to normal. You have to watch this, because if you use `OVER 1` in a program and don't cancel it with `OVER 0` later, you can still get overprinting taking place when you use another program unless you have switched off the QL between programs.

Just in case you think that normal is pretty dull, try the program in Fig. 8.10. This prints a phrase, pauses, then uses the command `STRIP 0` before `OVER 0`. The effect is the normal `OVER 0`, with the new letters replacing the old, but the difference is that the background is of a different colour. The black background is specified by the number that follows `STRIP`, and we shall be looking more closely at these colour numbers in a moment. `STRIP` allows you another way of drawing attention to something on the screen.

The `UNDER` command exists only in two forms, `UNDER 1` and

```

10 CLS
20 AT 2,4:PRINT"ANOTHER OVER 0 TEST"
30 PAUSE 100
40 STRIP 0
50 OVER 0
60 AT 2,4:PRINT"IIIIIIII"

```

Fig. 8.10. Highlighting text with STRIP.

UNDER 0. It allows you to underline text as it is being printed. To do so, the command UNDER 1 must come immediately before the piece of text that you want to underline. If this text is in the middle of a sentence, then you will have to use UNDER 0 to turn off the underlining following the piece that you wanted underlined. This will mean that you have to break up your PRINT statements carefully, using the semicolon to keep the printing in the correct line. Fig. 8.11 shows an example which you can use to base your own requirements on.

```

10 CLS:UNDER 1
20 AT 16,2:PRINT"TITLE"
30 UNDER 0
40 AT 2,4:PRINT"This shows how ";:UND
ER 1:PRINT"under 1";
50 UNDER 0:PRINT" can be used"

```

Fig. 8.11. How text on the screen can be underlined.

Write it in colour

It's time now to look at the colour instructions of QL. There are three particularly important ones, which use the instruction words BORDER, PAPER and INK. We'll start with BORDER. Now you have met the BORDER command already when you used it to place a border around a window. The first number which you use with BORDER will, as before, control the size of the border. The second number, following the comma, specifies the colour. You can also use a channel number in front of both if you are specifying a border for a window. When you are using the ordinary screen, as distinct from a window that you have specified, this border will be *within the ordinary border area*; it does not replace it. If you create a border, then any PRINT will always be to the screen inside the border. You can,

however, create a wide border in one colour, then a narrow border in another colour, so that a PRINT will be placed on the border area that was first printed! Using a border implies that you have to be careful how you lay out your work! Figure 8.12 shows an example, using two sets of loops which create borders of different sizes and colours. You can see

```

10 CLS
20 FOR n=0 TO 7
30 BORDER 5*n,n
40 PAUSE 100
50 NEXT n
60 FOR n=7 TO 0 STEP -1
70 BORDER 5*n,n
80 PAUSE 100
90 NEXT n
100 AT 7,5
110 PRINT"How's that for colours?"

```

Fig. 8.12. Creating borders in different colours.

that as the BORDER numbers increase, each border overprints the previous one, so that you see a border which changes colour and gets larger. Things look very different when the loop goes in the other direction, though. This time, each border remains as a strip in a different colour, and the overall effect is an interesting multicoloured border. If you PRINT anything now, you will have to use AT to make sure that it is placed in the 'screen' part rather than on the one of the inner BORDER parts.

The ordinary set of colour numbers that you can use with BORDER are shown in Figure 8.13. These numbers run from 0 to 7. If you use

Number	Colour
0	Black
1	Blue
2	Red
3	Magenta (red + blue light)
4	Green
5	Cyan (blue + green light)
6	Yellow
7	White

Fig. 8.13. The QL colour numbers. These can be used with BORDER, and with other commands, detailed in Fig. 8.14.

higher numbers (with a TV receiver), the colours will appear to have a sort of 'wire grid' over them, but with no noticeable change to the actual colours. Numbers above 128 give very striking effects, which you aren't supposed to try when you are using a TV receiver. Nothing went wrong with my receiver when I tried all of these numbers, but some numbers gave unpleasant rippling effects, and many caused the screen to show a pattern of vertical lines.

The next items are PAPER and INK. The names tell all, really. PAPER is used to select the colour of the background, and INK to select the colour of the letters or other shapes that you print on the paper. The range of colours is 0 to 7, as before. Figure 8.14 illustrates PAPER and INK in use. An instruction such as PAPER 6 does not, by itself, cause colour to appear. If we print on the screen following a PAPER 6 instruction, the background for our printing will appear in yellow (using a TV receiver), but only for the part on which we have printed. To make PAPER 6 colour a complete screen yellow, we have to follow it with a CLS instruction. That's illustrated in line 40 of Fig. 8.14. The second part of the program uses PAPER 0, black, and prints in different INK colours. You can see from the results of this program that if you want to keep your text clear, you have to use contrasting colours. Numbers that are very close to each other will never give enough contrast to make a good display. My own preference is to have the PAPER colour dark, and the INK colour light, because the opposite, a white screen with black print, can appear to flicker very irritatingly.

```

10 CLS
20 FOR N=0 TO 7
30 PAPER N
40 CLS
50 AT 10,12
60 PRINT"THIS IS PAPER ";N
70 PAUSE 100
80 NEXT N
90 PAPER 0:CLS
100 FOR N=0 TO 7
110 INK N
120 PRINT"THIS IS INK ";N
130 PAUSE 100
140 NEXT N

```

Fig. 8.14. PAPER and INK in use.

Don't expect the letters to appear in very distinct colours, because colour TV sets are not very good at displaying colour in small chunks. Add to that the fact that 90% of the male population is partially colour blind, and you'll see that the most impressive colour displays are the ones that use strong colours in big areas. Using INK with high numbers like 255 will produce *very* strange results when you use a TV receiver in place of a monitor. Anything else? Yes, one small point. If you change the PAPER colour, then any STRIP colour which you have been using will automatically be set to the new PAPER colour. If you want to use OVER 0 in any subsequent line, you'll have to specify the colour for STRIP again.

Odds and ends

Up to three numbers can be specified for a colour, but this is of interest only if you are using a monitor. I couldn't, at the time of writing, test the QL with a colour monitor, and I can't comment on how these colours might look. There are a few other assorted commands which are of interest in setting up print on the screen, but which aren't really important enough to illustrate in detail. One of these is CLS, with a number following it. The number allows you to select part of the screen for clearing, and the numbers are used in a way that is similar to their use in the SCROLL command. The Manual shows clearly what the actions are.

BLOCK is a command that has a specialised use in the creation of bar charts. Once again, it is well illustrated in the Manual, and requires no further illustration. SCALE is a remarkable command which allows you to vary the size of displays, and we'll look at it in much more detail in the next chapter. You have already met the use of CSIZE for altering the size of the QL characters.

Chapter Nine

Guidance with Graphics

The graphics abilities of the QL are very much greater than those of the ZX series of computers, including the Spectrum. 'Graphics' means the ability to make patterns and drawings on the screen, and this involves the use of commands which are very different from the commands that we use with text. One concept that we need to take a look at is 'resolution'. The resolution of a graphics display means the number of distinct pieces that would be needed to fill the screen. If you look closely at the screen of a colour TV, you will see that it is divided into a set of lines or dots. These are the bits that glow and give out light, and you can't have anything displayed on the screen which is smaller than one screen dot or the distance across a line. In fact, unless you use a monitor, you can never get anywhere near displaying dots so small or lines so narrow.

The 'dots' that the computer can work with are called *pixels* (*picture elements*). Each pixel that a computer can control corresponds to the size of several dots on the TV screen. The QL allows you two choices of resolution. The 'low' resolution is 256 pixels across by 256 down, a total of 65536 pixels which can be in any of eight colours (including black and white as separate colours). A lot of computers would call this 'high resolution', and it's as high as you could use along with a TV receiver. The 'high resolution' of the QL uses 512 pixels across by 256 down, a total of 131072 pixels. This is a very high resolution by any standard, and the price which has to be paid is a restricted number of colours, just black, red and green. In addition, you will need a monitor to make effective use of this resolution.

You can switch from one resolution to another by use of the MODE command. If you have pressed the F2 key when you switched on, your QL will be in 'low resolution' mode. You can change to high resolution by typing MODE 4 or MODE 512. When the ENTER key is pressed, you will see the screen clear, and if you attempt to print something on

the screen, you will see that the characters are now much smaller. To change back to low resolution, type `MODE 8` or `MODE 256`.

Making circles

The ability to 'paint' each of 65536 dots would not, by itself, be very useful if the only way to make pictures were to specify the position and colour of each pixel. There *are* computers which allow no other way of going about high resolution graphics, but the QL, as you might expect, does rather better. There is, in fact, an excellent variety of graphics commands which allow you to draw in a more sensible way. All of these commands make use of the X and Y co-ordinate system that we have already come across. The difference now is the numbers that you can use. For all of the graphics instructions, unless you alter the scale (using the `SCALE` command), the range of Y values is up to 100, and the range of X values is approximately up to 166 (Fig. 9.1). This last figure depends on how your TV receiver is adjusted. The important point is that equal numbers will cause equal distances to be covered. Five graphics steps in the X-direction is the same distance as five graphics steps in the Y-direction. This is true no matter whether you are using low or high resolution. In this way, any program that you have for creating graphics will work as well with either high or low resolution, just by changing `MODE`, assuming that you are using a monitor so as to be able to get the best from the high resolution mode. The other important point about these graphics co-ordinates is that they have a *different starting point*. Their origin is the *bottom* left-hand corner of the screen. The Y-distances are measured *upwards*, and the X-distances across. The direction for the Y co-ordinates is the opposite of the one that is used for the `CURSOR` co-ordinates. We'll start exploring graphics, then, by looking at the `CIRCLE` command.

Figure 9.2 shows a circle-drawing program. Line 10 sets the mode, just in case it had been changed, and clears the screen. I have made no attempt to change the `PAPER` colour. Line 20 then starts a loop, using numbers which will be the radii of the circles. Line 30 sets `INK` colours at random, which means that some circles will be in the paper colour and so will be invisible. Line 40 then draws the circles, and that's what we need to examine. There are three numbers following `CIRCLE`. The first two are the X and Y numbers for the centre of the circle. We want this to be the centre of the screen, so we take half of the permitted range of numbers in each direction. This means 83 for X and 50 for Y. The third figure is the radius (distance from the centre to the rim, in case

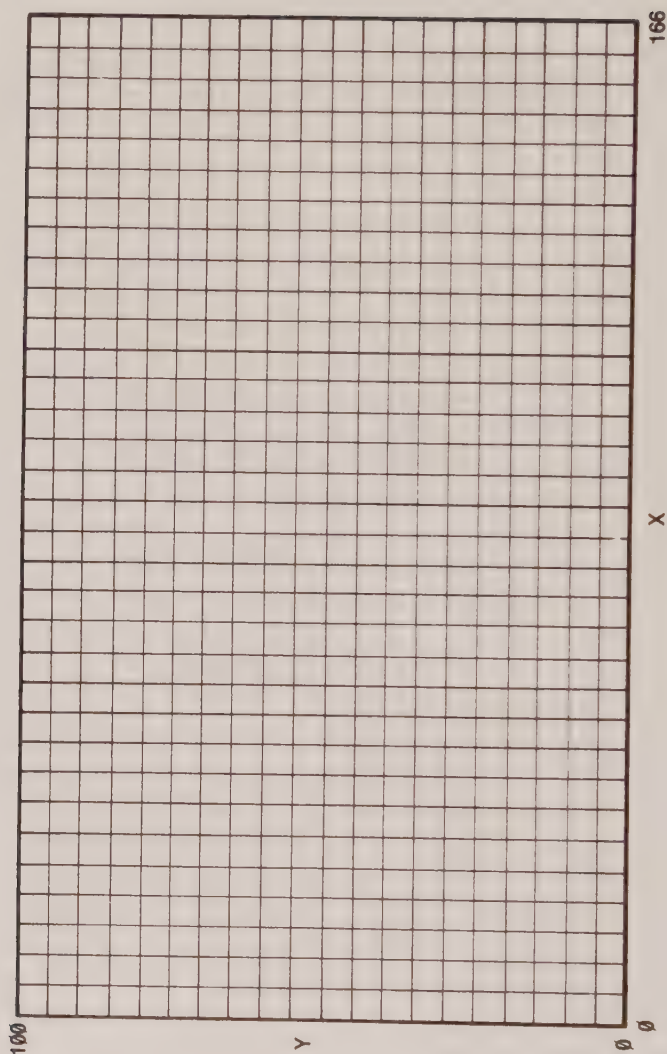


Fig. 9.1. The 'default' graphics co-ordinates. These can be changed by using SCALE. Note that the origin is at the *bottom* left-hand corner.

```

10 MODE 256:CLS
20 FOR N=5 TO 50 STEP 5
30 INK RND (0,7)
40 CIRCLE 83,50,N
50 PAUSE 100
60 NEXT N

```

FIG. 9.2

Fig. 9.2. A circle-drawing program, using CIRCLE.

you've forgotten). This is the variable N, so that the effect of the command will be to draw a set of circles, each of greater diameter than the previous one, and in randomly selected colours.

While we are working with the CIRCLE command, it's a good time to take a quick look at the FILL command. FILL means 'fill with INK colour', and this command has to be issued just before you draw a closed shape, like a circle or square. The correct form is FILL 1 (remember the space!); if you forget to specify an INK colour, then the command will use whatever ink colour has been used up till then. Figure 9.3 gives you a taste of this, with lines 10 to 30 drawing a circle and filling it with yellow ink. Each time you want to use a colour fill,

```
10 MODE 256:CLS
20 INK 6:FILL 1
30 CIRCLE 83,50,30
40 PAUSE 100
50 INK 3:FILL 1
60 CIRCLE 83,50,45
70 INK 6:FILL 0
80 CIRCLE 83,50,20
```

Fig. 9.3. Filling a closed shape with colour.

you have to use FILL 1 again, though you can turn the fill off by using FILL 0 if, for example, you don't want to have the command used. Lines 50 to 80 demonstrate this. As they stand, they will draw you a blue circle with a yellow circle inside it. If you omit the FILL 1 in line 70, however, you get an odd effect, a filled semicircle with a filled portion connecting this to the edge of the larger circle. To draw a yellow circle with no filling, you *must* use FILL 0 here. The manual doesn't make this very clear.

Squashing the circles

The CIRCLE command is capable of a lot more than the simple drawing of circles. One feature is that you can draw a whole set of circles with one CIRCLE command. Each set of numbers has to be separated by a semicolon, and this is illustrated in Fig. 9.4, with an

```
10 MODE 256:INK 6
20 CIRCLE 22,30,20;52,40,20;82,30,20;
112,40,20;142,30,20
```

Fig. 9.4. Drawing several circles with one CIRCLE command.

appropriate shape for 1984. More seriously, the **CIRCLE** command can be used to draw shapes that aren't circles! These shapes are ellipses, the sort of shapes that you can draw with a loop of thread, two pins and a pencil (Fig. 9.5). An ellipse has a centre, but no single radius value.

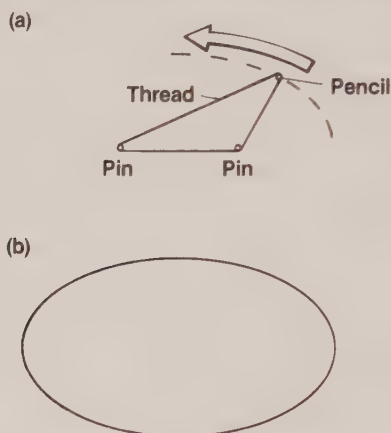


Fig. 9.5. Drawing with two pins, thread and a pencil (a). If the thread is kept taut, the result is an ellipse (b).

Instead, we use the measurements of the major and minor axes. The major axis of the ellipse is its greatest 'diameter', and the minor axis is its least 'diameter'. We also have to be able to specify whether the major axis is horizontal, vertical or at any other angle to the Y-direction. All of this needs another two numbers to specify. Just *two* numbers? Yes, because if we start the command by specifying a circle, we can convert this into an ellipse simply by specifying the ratio of major to minor axes. That's one number, and the angle to the vertical is another.

Figure 9.6 illustrates the effect of the 'eccentricity' figure, which is the ratio of major to minor axis. In the first part of the program, with the angle to the vertical zero radians, the effect of the eccentricity number

```
10 MODE 256: INK 4
20 FOR E=.1 TO 1 STEP .1
30 CIRCLE 83,50,30,E,0
40 PAUSE 50
50 NEXT E
60 PAUSE 150: INK 5
70 FOR E=1 TO 2 STEP .1
80 CIRCLE 83,50,30,E,0
90 PAUSE 50
100 NEXT E
```

Fig. 9.6. The effect of the 'eccentricity' number in the **CIRCLE** command.

rising from .1 to 1 is to draw ellipses which start very narrow, but become fatter as the number rises, until an eccentricity number of 1 produces a circle again. In the second part of the program, the eccentricity number rises from 1 to 2 in steps of .1, and this widens the shape further.

Figure 9.7 shows the effect of the angle number. This angle is measured starting with the vertical direction on the screen. Its units are

```
10 MODE 256:PAPER 0:INK 6:CLS
20 FOR A=0 TO PI STEP .1
40 CIRCLE 83,50,40,.5,A
50 PAUSE 20
60 NEXT A
```

Fig. 9.7. Altering the ANGLE number.

radians, rather than the degrees which will probably be more familiar to you. The radian is a more 'natural' way of measuring angle, which uses the number PI (π). The relationship between these methods is that PI radians (i.e. 3.142 radians) equals 180 degrees, but we don't have to worry about this very much except to visualise what effect an instruction will have. The QL, as you know by now, will accept the variable name of PI as meaning the number pi, and this is illustrated in the listing. Line 10 sets the PAPER and INK colours – note the need for CLS to set the PAPER colour. Line 20 uses angles 0 to PI, which means that the angle will change from 0 to 180 degrees. The step is .1 radians. An ellipse with eccentricity number of .5 is drawn at each angle, and the program pauses briefly between ellipses so that you can see the effect. A minor change to this will give you a rotating ellipse (Fig. 9.8). This time, the ellipse is drawn in INK 6, and then again in INK 0, making it appear, then disappear. The effect of doing this in each position is to create the illusion of a rotating ellipse.

The manual claims that another command, CIRCLE_R can be used with relative co-ordinates. Relative co-ordinates means that the co-ordinate numbers are measured from the cursor position, wherever

```
10 MODE 256:PAPER 0:CLS
20 FOR A=0 TO PI STEP .1
25 INK 6
40 CIRCLE 83,50,40,.5,A
52 INK 0
54 CIRCLE 83,50,40,.5,A
60 NEXT A
```

Fig. 9.8. Animation carried out by drawing in INK6 and in INK0 alternately.

that happens to be, instead of from the graphics origin at the bottom left-hand corner. The `CIRCLE_R` command produced a 'bad name' error message on my QL, however, so it may be that this command has been abandoned. Other commands which used relative co-ordinates appeared to work, however, and they will be dealt with as we meet them.

Arc at this

The ability to draw circles and ellipses is useful, but we often need to draw a section of a circle, or *arc*. The QL provides for this with the `ARC` and `ARC_R` commands. Taking `ARC` first, this requires two sets of co-ordinates and an angle. The co-ordinates are for the starting point of the arc and its ending point, and the angle is the angle through which the cursor has to turn as it draws the arc. If it didn't turn at all, it would trace a straight line, and if it turned through π radians, then the arc would be a semicircle. As you have gathered from that, the angle has to be in radians for this command. If you are still not keen on the idea of radians, then remember that the QL can carry out a conversion for you. You can, for example, use `RAD(180)` in place of π for 180 degrees.

Figure 9.9 shows a little of what can be done with `ARC`. The first set of arcs joins up a set of points which are spaced ten units apart, using `RAD(180)` as the angle so that the arc is semicircular. Now it should be possible to draw less curved arcs, using smaller angles, but the machine was rather inconsistent about this. As the listing shows, an angle of 60 degrees worked well, but angles greater than this and less than 180 degrees gave peculiar effects. If you have start-point and end-point in line, and you use an angle of between 60 and 180 degrees, then the arc is incomplete, which rather limits your artistic range. Angles greater than

```

10 REMark 9.9
20 CLS
30 FOR N=5 TO 155 STEP 10
40 ARC N,30 TO N+10,30,RAD(180)
50 NEXT N
60 FOR N=5 TO 155 STEP 20
70 ARC N,50 TO N+20,50,RAD(300)
80 ARC N,80 TO N+20,80,RAD(60)
90 NEXT N

```

Fig. 9.9. Using `ARC`. You have to be careful of the range of angles that you use.

180 degrees can be used, however, as line 70 illustrates. The ARC command can, in fact, be shortened if the cursor position can be taken as the new starting point. In such a case, a line like line 40 can be shortened to:

```
ARC TO N+10,30,RAD(180)
```

The ARC_R command can be used to give relative plotting. Wherever the graphics cursor happens to be at, the beginning of a command is taken as the origin for the co-ordinates. For example, if the cursor is at the point 83,50 then a point whose co-ordinates are 10,10 will actually be at 93,60, which is $83+10$, $50+10$. Using relative co-ordinates, you can have positions like -20,5 or 10,-20. Figure 9.10 shows ARC_R in use. Line 20 uses ARC simply to set a position for the cursor – though we'll look at another method later. The first loop then uses ARC_R to draw a set of connected arcs. After the pause, using negative co-ordinates draws another set of arcs in the opposite direction, producing the effect of loops which you can see. As with the CIRCLE command, you can draw more than one arc in a command line, providing the sets of commands are separated by semicolons.

```
10 CLS
20 ARC 5,10 TO 15,10,RAD(60)
30 FOR N=1 TO 8
40 ARC_R 0,0 TO 10,10,RAD(60)
50 NEXT N
60 PAUSE 100
70 FOR N=1 TO 8
80 ARC_R 0,0 TO -10,-10,RAD(60)
90 NEXT N
```

Fig. 9.10. ARC_R in use to make a series of arcs.

Drawing the line

I mentioned moving the graphics cursor, and this looks like a good place to deal with one of the commands that moves the cursor. The LINE command is one that takes three different forms, and the first form is one in which LINE is followed by two co-ordinate numbers. This has the effect of moving the graphics cursor to that position, but without making any mark on the screen. LINE 83,50, for example, will put the graphics cursor to the centre of the (TV) screen. You can use this to place the cursor ready for an ARC command.

LINE is more often used, as the name suggests, to draw a straight line from one point to another. If you use the form `LINE 30,50 TO 40,20`, then the line will be drawn between these two positions, using *absolute* co-ordinates (origin at the bottom left-hand corner of the screen). If you miss out the first pair of co-ordinate numbers, so that the command starts `LINE TO`, then the line is drawn from wherever the cursor happens to be to the point that is specified by the numbers which follow the word `TO`. Figure 9.11 shows `LINE` in use, and also shows how several lines can be drawn in one command by using the word `TO` between sets of co-ordinate numbers. `LINE_R` can be used to draw

```
10 CLS
20 LINE 83,90 TO 10,50
30 LINE TO 83,10 TO 160,50 TO 83,90
```

Fig. 9.11. `LINE` used for straight line drawing.

lines relative to the cursor position. Figure 9.12 shows a set of lines drawn from the centre of the screen to random positions, using `LINE` (line 50). After a pause, a quite different effect is achieved using `LINE_R`. This time, each new line is attached to the end of the previous one. By using line 110 to generate a value of `A` which is either `-1` or `+1`, the direction of each line is made random, and its size is also made to be random by line 120. This produces a pattern which is called a 'random walk' - you might like to think of it as the path of a demented fly.

```
10 CLS
20 LINE 83,50
30 FOR N=1 TO 50
40 INK RND(0,7)
50 LINE 83,50 TO RND(10,150),RND(10,90)
60 NEXT N
70 PAUSE 200:PAPER 0:CLS
80 LINE 83,50
90 FOR N=1 TO 50
100 INK RND(1,7)
110 A=RND(-0,1):IF A=0 THEN A=-1
120 LINE_R 0,0 TO A*RND(1,10),A*RND(1,10)
130 NEXT N
```

Fig. 9.12. `LINE` and `LINE_R` used in a random-line program. Note the difference.

Turtle topics

The method of plotting that uses X and Y numbers has some disadvantages. Suppose, for example, that you wanted to produce a square, and then draw a set of squares of different sizes. With X,Y graphics, you need to calculate a new set of X and Y numbers for each different square. There's another way of drawing patterns which gets around this limitation, and that's what we're going to look at now.

Suppose you defined a square this way. Take a point. Draw a line of 'K' points in length, upwards. Turn your direction of drawing through 90 degrees and repeat the line. Turn again through 90 degrees, and draw the line again. Do another turn, another line. That makes a square, because all squares have four 90 degree angles. If you want a small square, make the value of 'K' small; if you want a large square, make the value of 'K' large. That's all there is to it. Here's another advantage. Suppose that you want to draw the square tilted. All you have to do is to draw the *first line* at a different angle. The rest of the lines follow the same pattern of four 90 degree angles. It's a different method of thinking about drawings. This system is sometimes called *polar co-ordinates*; the X-and-Y method is called *Cartesian co-ordinates*. Another name for this new method is *turtle graphics*, because the commands are often used in conjunction with a remote-controlled 'turtle', which is a small motor-driven box which contains a pen. By issuing instructions, the box can be made to move, turn, lift the pen from the paper, or draw. We'll start work on this graphics system by looking at three important instructions, MOVE, TURN and PEN.

Your turn to turn turtle

Let's dispose of the PEN commands. PENUP means that the (imaginary) turtle should not mark the paper. This 'paper' is, in fact, the screen. PENDOWN is the opposite command, and after this, any command that moves the 'turtle' will leave a trail on the screen. There are only two other commands, TURN and MOVE. TURN means exactly what it says, turn the direction of the turtle. When you start from scratch, the turtle is always pointing to the right-hand side of the screen, and this direction is taken as zero degrees. Yes, degrees, because QL turtles don't use radians. If your turtle has become dizzy, and you want to set it to the correct starting direction, you can use the command `TURNTO 0` to make sure that it starts off correctly in this direction. This is the direction in which the turtle will move if you issue a MOVE

command. If you want it to point in any other direction, then you can use either TURN or TURNTO. Each command has to be followed by an angle in degrees. If you want to turn clockwise, the angle has to be *negative*; a positive number will turn it *anticlockwise*. TURNTO is an *absolute* command, meaning that all angles are measured with respect to angle 0, the horizontal-to-the-right direction. Using TURN allows you to use relative angles, measured from the direction that the turtle happens to be placed in.

It's time for a very simple example. How do we draw two vertical parallel lines on the screen? As it happens, that's the sort of thing that's more easily done using LINE, but we'll try it with the turtle. Figure 9.13 shows how this is done. Line 20 positions the cursor at 20,10, so that's

```

10 CLS
20 LINE 20,10
30 PENDOWN
40 TURNTO 90
50 MOVE 20
60 PENUP
70 TURNTO 0
80 MOVE 20
90 TURN -90
100 PENDOWN
110 MOVE 20

```

Fig. 9.13. Drawing two vertical parallel lines, using the 'turtle' commands.

where the turtle starts. Line 30 ensures that a line will be drawn, and line 40 makes the turtle point vertically up the screen. It's vertically *up*, not down, because 90 is a positive angle, which turns the direction of the turtle through 90 degrees anticlockwise. Line 50 then makes the turtle draw a line as it moves 20 steps up the screen. Line 60 stops the drawing, and line 70 makes the turtle point right again, using the absolute command TURNTO 0. We could just have easily have used TURN -90 here. Line 80 then causes the turtle to move 20 steps right, leaving no mark. Line 90 makes the turtle face downwards (a clockwise turn), and line 100 puts the pen down to make a mark so that MOVE 20 will draw another line. That's your two parallel lines!

Now if all of that looked rather tedious, which it is, it's because turtle graphics is not really suited to that type of drawing. Where it comes into its own is in making elaborate drawings, particularly geometrical patterns. Take a look at Fig. 9.14, which shows the speed and simplicity of this method of creating graphics. Lines 30 to 70 create a twelve-sided shape. This is done by specifying a loop of twelve passes, and using an

```

10 CLS
20 LINE 30,50
30 PENDOWN
40 FOR N=1 TO 12
50 MOVE 10
60 TURN 30
70 NEXT N
80 PAUSE 200
90 CLS
100 LINE 83,50
110 TURN RND(0,360)
120 FOR N=1 TO 100
130 MOVE RND(1,20)
140 TURN RND(1,270)
150 NEXT N

```

Fig. 9.14. Turtle graphics being used to create a geometrical pattern.

angle of turn which is $360/12$ degrees (i.e. 30 degrees). In the second part, turtle graphics are used to make a random pattern. Some of this may be off the screen, but this does not cause any error messages. You can make use of the MOVE and TURN commands to create new commands (using PROCedures) for any variety of the most popular turtle language, LOGO.

Lay it on the line

Before we continue to explore the graphics abilities of the QL, another few words on planning may be useful. The more elaborate your graphics become, the more planning they need, but the process of planning is not necessarily more difficult. The essential part is to be able to locate points on your plan, and so it's very important to make your plans either on graph paper, or on tracing paper clipped over graph paper. No matter which type of graphics you use, you will always have to plot points, and when you plan your drawings on graph paper, it's much easier to see the shape and scale of the result than if you just 'think of a number'. Unplanned graphics can be fun, but they can also waste a lot of time and patience.

Pointing it out

There's one very important aspect of graphics which none of these

commands deals with, drawing graphs. A graph is a set of points which can be joined and which should convey some information. The QL will plot graph points for you, using the POINT or POINT_R commands. As usual, the POINT command uses absolute co-ordinates and the POINT_R command uses co-ordinates measured relative to the cursor position. The colour of the point that is plotted will be whatever INK colour happens to be in use at the time.

Figure 9.15 shows the QL being used to plot three graphs at the same time, and doing so at a reasonable speed. The range of X in line 20 is set

```

10 PAPER 7:CLS
20 FOR X=1 TO 165
30 Y=50+50*SIN(2*PI*X/165)
40 INK 1:POINT X,Y
50 Y=50+50*(SIN(2*PI*X/165)^2)
60 INK 2:POINT X,Y
70 Y=50+50*(SIN(2*PI*X/165)^3)
80 INK 3:POINT X,Y
90 NEXT X

```

Fig. 9.15. Plotting graphs, three at a time.

so as to cover the width of the screen, and for each value of X, three values of Y are calculated. One is obtained from the sine of the angle whose value is $X/165$, in radians. Another is obtained from the square of the sine of this angle, and the third is obtained from the cube of the sine of the angle. Each point is plotted by the POINT commands in lines 40, 60 and 80, using different INK colours for the three different graphs. The reason for the numbers that are used is that we want the graphs to fill the screen. The sine of an angle has a value that lies between -1 and $+1$. Now a value of 1 in the Y-direction does not make much impression, so we multiply the value by 50 . This makes the quantity vary between -50 and $+50$. We can't plot -50 , however, so we add 50 to each value, making the size vary between 0 and 100 , the full range of Y values. The other point is that we use $2*PI*X/165$ as the angle. This is to allow for radian measure. These angle functions go through a range of values as the angle goes from zero radians to $2*PI$ radians. When $X=0$, then $2*PI*X/165$ is zero, which gives us zero radians at the left-hand side of the graph. When $X=165$, at the right-hand side of the graph, then the angle is $2*PI*165/165$, which is $2*PI$ radians, just what we want.

Now, after running that program, try this. Type SCALE 500,0,0 and press ENTER. Now run the program again. This time, your graph is squeezed down in the left-hand corner of the screen! The SCALE

command does what the name suggests, it scales up or down all the graphics effects. By using `SCALE 500,0,0` we are making the screen behave as if the maximum Y number is now 500, and the maximum X number is altered in proportion (to 5×165 , approximately). The graph which we have drawn is therefore very small. If you use `SCALE 10,0,0` you won't see any of the graph at all, because none of it will fit into this portion! The use of `SCALE` makes graphing much easier, because we can forget the use of the scaling numbers that we illustrated in Fig. 9.15.

Take a look at Fig. 9.16. This uses `SCALE 2,0,-1`. The first number gives the maximum number which represents the full vertical height of the screen. The next two give the position that will be represented by the bottom left-hand corner of the screen. By choosing 2 as the vertical size,

```
10 PAPER 7:CLS
20 SCALE 2,0,-1
30 FOR N=0 TO 3 STEP .1
40 POINT N,SIN(2*PI*N/3)
50 NEXT N
```

Fig. 9.16. Using `SCALE` for a graph plotting program.

we allow for the value of the sine of the angle to go from -1 to $+1$. By making the bottom left-hand corner of the screen the point $0,-1$, we ensure that when the value of the sine is -1 , it will be plotted at the bottom of the screen. In this way, we also ensure that $+1$ is plotted at the top, because the range of vertical values is 2. The X number is left at 0, because we want to plot angles from zero to $2 \times \text{PI}$ as before. The use of `SCALE`, however, makes the expressions much simpler. Remember, however, that you need to carry out `SCALE 100,0,0` to get the graphics screen back to normal again afterwards.

There's another command, `RECOL`, which affects any of our graphics or print instructions. `RECOL` changes all of the colours on the screen. The command `RECOL` has to be followed by eight numbers. Each of these numbers specifies the new colour that you want to use in place of an old one. The colours are listed in the order, blue, red, magenta, green, cyan, yellow, white and black. If, for example, you wanted all the screen colours to be black, you would use `RECOL 0,0,0,0,0,0,0,0`. This isn't quite as daft as it sounds, because after issuing this command, you could draw a pattern on the screen invisibly. You could then make it appear by using `RECOL 1,2,3,4,5,6,7,0`. You could also have a set of drawings made in different colours while the screen was black, and you could issue a set of `RECOL` instructions which revealed *one drawing at a time* in whatever colour you wanted to use. This is a very useful way of achieving animation without too much effort.

Chapter Ten

Sound Sense

The ability to produce sound is an essential feature of all modern computers. The sound of the QL comes from a built-in loudspeaker which is considerably better than the type which is fitted to the Spectrum. You have, however, no apparent control over the volume of sound. In addition, the range of available notes is rather restricted when compared to other machines.

What we call sound is the result of rapid changes of the pressure of the air round our ears. We don't notice these pressure changes unless they are fairly fast, and we measure the rate in terms of cycles per second, or hertz. A cycle of a wave is a set of changes of pressure, first in one direction, then in the other and back to normal, which we can illustrate by the graph in Fig. 10.1. The reason that we talk about a sound 'wave' is because the shape of this graph is a wave shape.

The *frequency* of sound is its number of hertz – the number of cycles of changing air pressure per second. If this amount is less than about 20

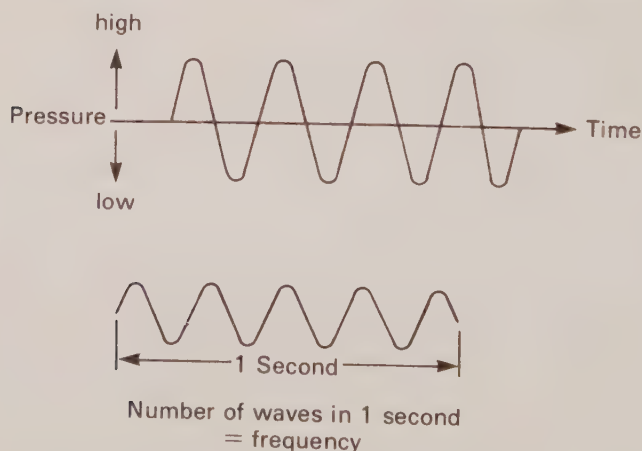


Fig. 10.1. Sound waveforms, showing how the air pressure changes with time. The number of changes per second is called the frequency.

hertz, we simply can't hear it, though it can still have physically disturbing effects. We can hear the effect of pressure waves in the air at frequencies above 20 hertz, going up to about 15000 hertz. The frequency of the waves corresponds to what we sense as the 'pitch' of a note. A low frequency of 80 to 120 hertz corresponds to a low-pitch bass note. A frequency of 400 or above corresponds to a high pitch treble note.

The amount of pressure change determines what we call the loudness of a note. This is measured in terms of *amplitude*, which is the maximum change of pressure of the air from its normal value. For complete control over the generation of sound, we need to be able to specify the amplitude, frequency, shape of wave, and also the way that the amplitude of the note changes during the time when it sounds.

For the purposes of writing music, a composer has to specify for each note how *loud* it will be, for how *long* it has to be played, and its *pitch*. In written music, loudness is indicated by letters such as *f* (loud) and *p* (soft), and by using more than one letter if necessary. For example, *fff* means very loud, and *ppp* means very soft. The duration of a note is indicated in two ways. One of these ways is a metronome reading. A metronome is a sound generator which ticks at precise intervals, and a metronome reading indicates how many 'beats' (units of notes) are sounded per minute. The unit duration of note is called the *crochet*, so the metronome reading decides on the time of a *crochet* by measuring the number of *crochets* that would be sounded per minute. The durations of the other notes are then measured in comparison to this. A *minim* sounds for twice as long as a *crochet*, and a *semibreve* sounds for twice as long as a *minim*, which is four times as long as a *crochet*. The *quaver* is allowed half the time of a *crochet*, and a *semiquaver* a quarter the time of the *crochet*. These differently timed notes are indicated by the shape of the symbols that are used for the notes (Fig. 10.2). In addition, there are symbols for different durations of silence in the

Symbol	Time	Name
	$\frac{1}{8}$	Demisemiquaver
	$\frac{1}{4}$	Semiquaver
	$\frac{1}{2}$	Quaver
	1	Crotchet
	2	Minim
	4	Semibreve

Fig. 10.2. The symbols that are used in written music to indicate the time of each note.

music, and these are illustrated in Fig. 10.3. Some music scores do not show a metronome reading, but rely on the use of Italian words to indicate the time of a crochet less precisely.






Rest Symbol	Time
	$\frac{1}{4}$
	$\frac{1}{2}$
	1
	2
	4

Fig. 10.3. The symbols for silences in written music.

The pitch of a note is indicated in written music by placing it on a kind of musical 'map' that is called the *stave*. Piano music shows two of these staves, each consisting of five lines and four spaces. The set of lines which is marked with the sign like the ampersand (&) is the *treble stave*, used for the higher notes, and the stave below it is the *bass stave*. The bass stave is also marked with a distinctive symbol, like a reversed 'C'. When music is written for instruments which do not use a keyboard, then only one stave is used. Piano and organ music always shows two staves, with a place for a note placed between them. This note is called *Middle C* and this note is played on a piano by pressing the key which is almost exactly at the centre of the keyboard. Figure 10.4 shows the staves with the notes marked.

The notes that are shown in Fig. 10.4 are arranged in groups of eight,

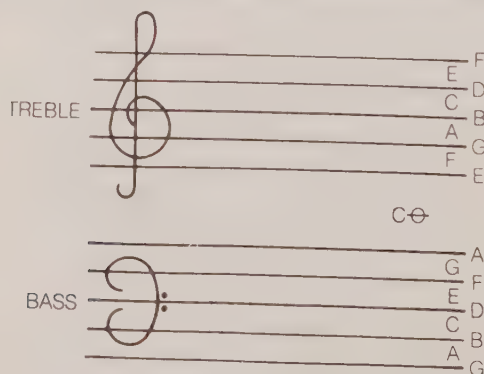


Fig. 10.4. The staves, with the names of the notes written in.

counting inclusively. This group is called an *octave*. Music from the Western hemisphere traditionally uses a total of twelve distinct notes – tones and semitones – in an octave, and this full range is illustrated in Fig. 10.5, which shows the appearance of part of the piano keyboard. The half-pitch notes, or *semitones* are marked on the piano mainly by the black keys, though two semitones (between B and C, and between E and F) are not marked in this way. On written music, semitones are indicated by using the signs # (sharp) and ♭ (flat). A sharp indicates that the pitch has to be raised one semitone above the written note, and a flat indicates that the pitch is to be lowered one semitone below the written note. On the piano, the semitone above one note is the same as the semitone below the next note, so that C# is the same as D♭. This is not necessarily true on other instruments.

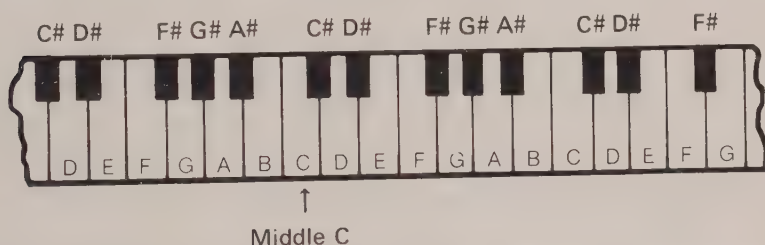


Fig. 10.5. Part of the piano keyboard, showing Middle C. There is only one semitone between B and C, and between E and F.

The QL sound

To work, then. The QL provides a sound command, BEEP, which can be used either as a simple command, or as a complicated one. How you use it depends on what you want of it, so that you don't have to do a lot of planning just to produce a warning note. When you are experienced in the use of the sound command, however, you can make use of it in much more interesting ways. We shall start with the straightforward production of notes by the BEEP instruction.

The BEEP instruction has to be followed by two numbers. Of these, the first number is a *duration number* which decides for how long a note will play. The range of numbers that you can use is 1 (too short to hear!) to 32767, which is a fairly long note. The number 0 has a special meaning – it will make a note sound until you cancel it. These duration numbers can be negative or positive; it's the absolute values which count.

The second number that follows the BEEP instruction is the *pitch*

number. This controls the pitch of the note that is produced by the QL, and its range is from 0 (highest pitch note) to 255 (lowest note). The closest musical equivalents are shown in Fig. 10.6, which shows the pitch numbers that correspond most closely to written notes and also to the piano scale. The piano is the most familiar type of musical instrument, and its keyboard is set out so as to make it very easy to play one particular series of eight notes, called the 'scale of C Major'. The scale starts on the note called Middle C, and ends on a note that is also called C, but which is the eighth note above Middle C. A group of eight notes like this covers an octave, so that the note you end with in this scale is the C which is one octave above Middle C. The BEEP command provides reasonably well for notes below Middle C, but not so well for notes above Middle C. This rather limits the use of this command for music.

Number	Note	Number	Note
11	C' (octave above Middle C)	35	B (below Middle C)
12-13	B	38	A#
14	A#	41	A
15	A	44	G#
17	G#	47	G
18	G	51	F#
20	F#	55	F
22	F	59	E
24	E	63	D#
26	D#	67	D
28	D	72	C#
31	C#	77	C (octave below Middle C)
33	Middle C		

Fig. 10.6. Pitch numbers for the QL. This list covers one octave on each side of Middle C.

Let's start our investigation of the BEEP instruction with a few single notes. These are illustrated in Fig. 10.7, which has three BEEP instructions, with PAUSE steps between them. These pauses are, as we shall see, very important. When the program runs, you will hear a short note of about Middle C pitch, then a long note of the C above Middle C, then a short note which is the C below Middle C.

Now the pauses are important if you want to hear these notes separately. Try removing line 25. When you run the program this time, you will hear the first note as before, but the second and third notes are


```

10 BEEP 2000,33
15 PAUSE 40
20 BEEP 20000,11
25 PAUSE 70
30 BEEP 1000,77

```

Fig. 10.7. Playing single notes with the BEEP instruction. The pauses are important.

sounded almost together, and only for the duration of the shorter note in line 30. If you now remove line 15, you will hear all three notes sound for a very brief interval as the program runs. What in fact is happening is that the first note sounds, then is killed when the second BEEP instruction starts, so that the second note takes over. This in turn is switched off by the third note. You can stop any note from sounding by using another BEEP command, even a BEEP with no numbers following it.

The next step is to try a musical scale. There is nothing in the QL Manual to show how the BEEP pitch numbers are related to musical notes, so you will have to rely on the scale of Fig. 10.6. Making use of this figure, we arrive at the program in Fig. 10.8, which plays the scale of C Major. This is a scale which starts with Middle C, and goes up to

```

10 FOR N=1 TO 8
20 READ PITCH
30 BEEP 20000,PITCH
40 PAUSE 60
50 NEXT N
100 DATA 33,28,24,22,18,15,12,11

```

Fig. 10.8. The scale of C Major, by QL. Some of the notes cannot be exactly obtained.

the C above it. In this scale of notes, the frequencies are such that the C above Middle C has exactly double the frequency of Middle C itself. The same is true of all the other notes in the scale – doubling the frequency is equivalent to going up one octave in pitch. Halving the frequency corresponds to going down one octave in pitch. The QL pitch numbers operate the opposite way round, so that a rise in pitch is achieved by reducing the QL number. You can see from Fig. 10.8 that Middle C uses a number of 33, and the C above it uses 11. There is no simple relationship between these BEEP numbers and the frequency numbers for these notes.

Unlike a lot of modern computers, the QL has only one channel of sound, so that you can't produce harmony. There is little point, then, in

looking closely at the composition of music, but a guide to producing simple melodies may be useful. Unless you are an accomplished musician, or want to be, it's best to use sheet music as a guide. The best music to use is cello or baritone voice music. If you want to use piano music, you will have to be certain which part of the score contains the melody – easy if you can read music, but not so easy if you don't. You will find that a lot of music requires high notes that you simply can't program on the QL. Avoid music for instruments such as the clarinet or bassoon, because these are 'transposing instruments'. This means that the notes which they sound are *not* the notes that appear in the written music!

The best technique to use is a loop for the number of notes and silences that you will use. The PAUSE which causes a silence should be programmed with a variable rather than a number, so that you can speed up or slow down the music without having to change every line. When you try this at first, it's advisable to have one line of DATA for each note. You'll find that piano music usually sounds better if you have short pauses between notes, but that organ music doesn't need this. Experience is the main thing that you need once you have acquired the programming skills.

```

5 RESTORE
10 S=7:R=2000
20 FOR J=1 TO 15
30 READ M,D,P
40 BEEP D*R,N
50 PAUSE P*S
60 NEXT J
100 DATA 33,4,4
110 DATA 33,2,2
120 DATA 33,2,2
130 DATA 11,6,6
140 DATA 12,2,2
150 DATA 12,2,2
160 DATA 15,6,9
170 DATA 15,2,2
180 DATA 18,4,4
190 DATA 24,4,4
200 DATA 24,4,3
210 DATA 28,2,2
220 DATA 33,2,2
230 DATA 24,8,8
240 DATA 28,4,4

```

Fig. 10.9. A simple tune, written from sheet music.

As an example, look at Fig. 10.9, which illustrates a simple tune. This was written with the aid of sheet music, which is comparatively straightforward, because the main program consists of a loop, with N,D and P used for the note, duration and pause numbers respectively. Using this form makes it easy to change the tune by altering the DATA lines. The actual duration and pause numbers are obtained by multiplying the numbers in the DATA lines by constants that are placed in line 10. I wrote the numbers for the DATA lines to produce what I thought would give round about the correct values for the tune. The original numbers which I used in line 10 caused the tune to be played very slowly, so I could adjust the DATA numbers if necessary. All I had to do after that was to juggle with the numbers in line 10 so as to get the tempo (the speed) of the music about right.

Special effects department

The BEEP instruction, even in its simple form, can produce a large range of useful sound effects. Let's start with a rising pitch of note which makes a useful warning, or a 'something about to happen' note. This is illustrated in Fig. 10.10. The loop that starts in line 10 uses values

```
10 FOR N=255 TO 0 STEP -1
20 BEEP 2000,N
30 NEXT N
```

Fig. 10.10. A warning note, of increasing pitch.

of N that range from 255 to 0, the full range that the BEEP instruction uses. These are the numbers that we shall use as pitch numbers in the BEEP instruction in line 20. The duration number has been fixed at 2000. This might seem rather long, but in fact, the loop does not run noticeably quicker with smaller values of duration number. More important, the use of smaller duration numbers gives much less satisfactory notes. You can shorten the time of this warning note by making the range less, perhaps 100 to 0 instead of 255 to zero.

Figure 10.11 shows a program that produces a warbling note. This is particularly useful for attracting attention, or for announcing an event in a game. For some reason, a warbling note attracts our attention more than a single note, which is why a warbling note was chosen for the later types of telephones. The warble in this program uses the loop that starts in line 10. This sounds 100 pairs of notes, which are short with a duration set by the short time delays in lines 30 and 50. The two

```
10 FOR N=1 TO 100
20 BEEP 2000,33
30 PAUSE 2
40 BEEP 2000,34
50 PAUSE 2
60 NEXT N
```

Fig. 10.11. A warbling note program.

pitch numbers that have been chosen in this example are 33 and 34. Higher pitches are even more effective, and values like 3 and 4 give effective attention-getting warbles. Note in this example again, that a duration number of 2000 is about the minimum that is needed. Using 1000 is distinctly less satisfactory, particularly if you use pitch numbers of 3 and 4. You may, of course, decide that you like the effect that a duration of 1000 with pitch numbers of 3 and 4 produces, since it seems at time to produce more than two notes!

BEEP extended

The BEEP command of the QL differs from the BEEP command of the Spectrum in allowing quite a lot more in the way of sound effects. These additional effects have to be programmed by adding other numbers to the BEEP command, following the duration and pitch numbers. These extra numbers affect what is called the *envelope* of the note. The effect is anything but simple, and because it's difficult to describe in words what a noise sounds like, you simply have to try the programs and listen!

First of all, however, I have to explain what is meant by an 'envelope'. The sounds that the simple BEEP command produces have a constant amplitude and constant frequency. In simpler words, their loudness is constant and so is their pitch as the notes play. Musical instruments, however, produce notes in which the loudness varies in each note. A piano note, for example, is loud at the instant it is struck, because that's when the hammer strikes the strings. This dies away rapidly, and it's the way in which the note's loudness dies away that makes a piano note so distinctive. Other instruments produce notes which behave quite differently, and all instruments produce notes which consist of a mixture of frequencies. That's why you can tell a piano from a violin from a flute from an oboe, even if they are all playing the same note. A graph of the amplitude of a note, plotted over

the time that the note takes, is called the 'envelope' of the note. The QL does not allow you control over this type of envelope.

For a lot of sound effects, however, we also want to be able to control the rate at which a note changes pitch. This is provided for in the BEEP command by adding four more numbers following the duration and pitch numbers. The first of these numbers is the second pitch. If we want the note to change pitch, we have to specify what it changes to. We also have to specify how fast it changes, and another number, the GRAD_X number does this. The GRAD_X number follows the second pitch number. For warbling notes, we will want the pitch to change back again, and the rate at which this happens is decided by another number, the GRAD_Y number. Finally, we have to specify how often we want this change to take place, and that's decided by another number, the 'wrap' number. Figure 10.12 summarises these commands, and shows the range of number that can be used for each. Note that the range of 'GRAD_X' is 0 to 15, but the range of 'GRAD_Y' is -8 to +7.

Pitch 2 note: Another pitch number. Sound can be made to change frequency between the first pitch number and this second one. Range 0 to 255.
Grad X: Specifies rate at which sound will change frequency from first pitch to second. Range 0 to 15.

Grad Y: Specifies rate at which sound will change frequency from second pitch to first. Range -8 to +7 (0 causes no effect).

Wrap: Specifies number of times that the sound will change between the two pitch numbers. Range 0 to 15.

Fuzzy: Causes noise to be added to the sound. Range 0 to 15.

Fig. 10.12. A summary of the extra number commands that can be added to BEEP.

Figure 10.13 gives you some idea of what is going by running through some combinations of 'GRAD_X' and 'GRAD_Y' numbers. High values of GRAD_X and GRAD_Y will hold the second pitch number for longer than the first; low values will do the opposite. If you

```
10 FOR N=0 TO 15
20 PRINT "GRAD_X IS ";N;"GRAD_Y IS "
   ;N-8
30 BEEP 32767,33,37,N,N-8,10
40 PAUSE 200
50 NEXT N
```

Fig. 10.13. Illustrating the effects of the extra 'GRAD' numbers.

make both values equal to zero, then a steady note is sounded. If you want to make use of the other two control numbers, but without specifying a rate of warbling, then you will have to use a 0 in each 'grad' position, and you must also specify a second pitch.

The next control number is referred to as the 'fuzzy' number. Its effect is to add noise to the sound. Noise is sound that has no single frequency, or group of frequencies which are related. It is a random mixture of frequencies in which a few frequencies may be louder than others. Lend an ear to the program of Fig. 10.14. This uses BEEP 0 to sound for a time which will be controlled by the PAUSE rather than by the duration number. The loop number, N, controls an extra seventh number in the BEEP line, and this is the 'fuzzy' number. You will find that the lower values of the 'fuzzy' number have no noticeable effect, but when you get to 8 and above, the effect is quite dramatic. As a way of generating noises, it's quite useful, and it can be combined with the changes of pitch that we used previously. An extra number, 'rand' can be added, with a range of values from -32767 to +32767, but I could not hear any effect from this number on any of the sounds that I tried.

```

10 CLS
30 FOR N=0 TO 15
35 PRINT N
40 BEEP 0,33,44,0,0,14,N
50 PAUSE 100
60 NEXT N
70 BEEP

```

Fig. 10.14. The effect of the 'fuzzy' number.

It's not easy to create convincing sound effects with BEEP, even the extended version. The effects like gunshots, drums, pneumatic drills, and so on that you hear from other computers are just not easily obtained from BEEP. I would not go so far as to say that it's impossible to get these effects, but it will certainly be necessary to experiment with different numbers for a much longer time than I could devote. For the simpler range of noises, however, the BEEP command is adequate.

Appendix A

Editing

Editing means changing something that has already appeared on the screen. Any feature of a line, *including its line number* can be changed by editing. The editing process can be carried out:

- (a) while a line is being entered, before ENTER has been pressed;
- (b) after pressing ENTER, if there is an error message and the line is not entered;
- (c) at a later stage, after a line has been entered, but before the program is run;
- (d) when an error is signalled during running.

Dealing with these in order:

- While a line is being entered, all of the editing commands, below, can be used. Editing is completed by pressing the ENTER key.
- If there is an error message after pressing ENTER, the line is not entered in the memory. You will have to retype the whole line in this case.
- When the line has been entered, but the program has not been run, you must prepare for editing by typing EDIT 200 (using the correct line number), and then pressing ENTER.
- When the program stops with an error message, the error message will show the number of the line in which the error has been detected. This does not always mean that there is an error in *that* line. For example, if the line contains a READ command, and the DATA is faulty, an error will be signalled in the line that contains the READ, not in the line that contains the DATA.

Editing commands

1. Cursor keys. The two keys on the left-hand side of the spacebar are

the main editing keys. When you bring down a line for editing (by a command such as EDIT 200), the cursor is at the end of the line. It can be moved only backwards from this position, using the left-arrow key. Once the cursor has moved from the end of the line, you can move it with either the left-arrow or right-arrow keys. If you hold these keys down, their action repeats very rapidly. The up-arrow and down-arrow keys have no effect.

2. To insert one or more letters, simply place the cursor over the *following* letter and type the new letter or letters.
3. To delete a letter, place the cursor over the following letter, and press CTRL left-arrow. Alternatively, you can place the cursor over the *preceding* letter and then press CTRL right-arrow.
4. The Manual lists a number of commands which use SHIFT and ALT along with the arrowed keys. None of these commands worked on my QL.
5. Pressing ENTER ends editing, and places the line into memory – unless another error is detected! If there is another error, the line will not have been changed in any way, and you will have to edit all over again.

Other related actions

1. Very often, it's useful to be able to delete a group of lines. You might, for example, want to construct a new program which uses a couple of PROCedures from an old one. You can remove a range of unwanted lines by using DLINE. For example, DLINE 10 TO 100 will remove all of the lines from 10 to 100 inclusive.
2. If you want to enter a long program, whose lines are numbered in an orderly way, you can use AUTO. Typing AUTO (then ENTER) gives you the line number 100, and each time you press ENTER, you will get a line number which is incremented by 10 (110, 120, 130 etc.). If you want, for example, lines starting at 10 and incrementing by 5, then type AUTO 10,5. To escape from auto-numbering, press CTRL and SPACE.

Appendix B

Using Printers

The QL, in its natural state, can be connected only to 'serial printers'. This means printers which take each item of data as ten or eleven (depending on design) signals, which must be sent in sequence along a cable. Most of the better-quality and lower-priced printers use what is called a *parallel interface*, which uses a set of cables to transmit one unit of data at a time. Some printers which normally come with a parallel interface can be fitted with a serial interface for about £60 or so extra. The listings in this book were printed with an Epson RX-80 which was fitted with its optional serial interface.

At the time of writing, one supplier was offering a parallel interface for the QL, and this was cheaper than a serial interface for most printers. If you have a serial printer, you will have to check the following:

1. Baud rate. This corresponds to the rate at which data can be sent. If your printer allows operation at 4800 Baud, then set it for this speed.
2. Number of data bits. Set this to 8.
3. Parity (a method of detecting errors). Set to 'no parity'.
4. The connections between the QL and the printer should permit 'handshake'; that is, the printer can signal to the QL when it is ready for data.
5. If your printer can be set up in this way (and most good-quality printers, such as the Epson range, can), then two commands to the QL should get you printing. These are:

BAUD 4800:OPEN#3,SER1

6. If your printer cannot be adjusted (like the Tandy colour printer/plotter) then you will have to add to the SER1 part of the command to allow for this. See the QL manual for details.
7. Once you have established the printer on channel #3, you can use

PRINT#3, LIST#3 etc. You must remember, however, that after using NEW, you will have to use OPEN#3, SER1 all over again, because a NEW resets all of the channels. This can be very annoying if you want to print a collection of programs, and I found it better to use DLINE to delete the lines of programs rather than using NEW.

Index

- # sign, 106
- & sign, 37
- ABS, 63
- adaptor, TV cable, 3
- adaptor, mains, 5
- ADATE, 83
- aerial lead, 3
- alphabetical order, 78
- altering file, 126
- ampersand (&) sign, 37
- amplitude, 159
- angle number, 149
- animation, 149
- apostrophe, 27
- ARC, 150
- arithmetic actions, 22
- array, 78
- ASCII code, 67
- assignment, 33
- asterisk, 23
- AT, 28
- AT map, 30
- AUTO, 170
- automatic line numbering, 170
- awkward fractions, 46
- backing up, 15
- backslash, 23
- backup, 13
- bad line message, 35
- bad name message, 12, 21
- BASIC, 20
- baud rate, 171
- becomes, use of =, 44
- BEEP, 161
- blank string, 66
- BLOCK, 143
- BOOT filename, 19
- BORDER, 133, 140
- break out of loop, 53
- breaking loop, 51
- buffer, 110, 112
- byte, 105
- cable, TV, 4
- CAPS LOCK key, 11
- care of cartridges, 18
- carriage return, 12
- Cartesian co-ordinates, 153
- cartridges, 5
- centring a title, 29, 70
- channels, 40, 104
- character size, 31
- characters, 69
- CHR\$, 75
- CHR\$(0) use, 118
- CIRCLE, 145
- CIRCLE numbers, 145
- CLEAR, 51, 126
- CLOSE, 112
- CLS, 17
- co-ordinate system, 130
- co-ordinates, graphics, 145
- CODE, 75
- coding messages, 75
- colon, 26
- colour instructions, 140
- colour numbers, 141
- colour receiver, 4
- column number for AT, 28
- comma, 27
- compare strings, 77
- comparing numbers, 61
- computer stand, 5, 6
- concatenation, 37
- connecting window to channel, 134
- COPY, 114
- copyright notice, 7
- core program, 94

- corrupted names, 127
- counting, 44
- counting characters in string, 69
- creating file, 109
- crochet, 159
- CSIZE, 31
- CTRL key, 11
- cursor, 10
- CURSOR, 137
- cursor keys, 12, 169
- damage to QL, 10
- damage to cartridges, 10
- data, 105
- DATA, 42
- data bits, 171
- database programs, 91
- DATE, 83
- DATES, 82
- DAYS, 83
- decrementing, 44
- defined function, 48
- delete letter, 170
- delete lines, 170
- designing programs, 91
- destination device, 114
- devices, 104
- dial tuning, 7
- DIM, 79
- dimension array, 79
- dimension string array, 80
- DIR, 17
- direct mode, 20
- DIV, 48
- DLINE, 170
- dollar sign, 67
- drawing graphs, 156
- duration number, BEEP, 161
- EASEL program, 14
- eccentricity, ellipse, 148
- editing commands, 20
- editing, 169
- ellipse, 148
- ELSE, 53
- end of file character, 118
- end of file message, 76, 117
- end-of-file signal, 113
- ENDFOR, 55
- ending condition, 52
- ENTER key, 12
- envelope, 166
- ESC key, 11
- exclamation mark, 36
- EXIT, 52
- expression, 33
- extension lead, 4
- F1 key, 7
- Fidelity MTV100, 3
- field, 107
- file, 107
- filename, 17
- FILL, 147
- FILL\$, 38
- finding names in file, 121
- fine-tuning, TV, 10
- FLASH, 137
- flat, 161
- FOR, 55
- forbidden operation, 37
- formatting, 15
- forward slash, 31
- frame for title, 38
- frequency, 158
- fuse, 1
- fuzzy number, 168
- game procedure, 96
- GOTO, 50, 63
- GRAD numbers, 167
- graph drawing, 156
- graphics abilities, 144
- graphics co-ordinates, 145
- graphics co-ordinates map, 146
- graphics cursor, 151
- graphics origin, 145
- graphics planning, 155
- handshake, 171
- hard copy, 25
- hashmark (#), 106
- IF, 52
- incrementing, 44
- indicator light, Microdrive, 10
- INK, 142
- INKEYS, 65, 120
- inner loop, 57
- INPUT, 38
- insert letters, 73, 170
- INSTR, 83
- instruction words, 12
- integer variable, 47
- intelligent spacer, 36
- inverted commas, 24
- keyboard, 66
- KEYROW, 66

KEYROW numbers, 65

leaving files open, 113

leaving loop early, 56

LEN, 69

length of string problem, 74

LET, 34

LINE, 151

line number, 22

line number for AT, 28

line numbers, 16

LIST, 17

LN, 45

load and run, 15, 18

loading, 13

local variable, 49

lock-up, 51

LOG10, 45

LOGO, 155

lower-case, 11

machine code, 91

magnetic disks, 13

magnetic tape, 13

mains plug connections, 2

mains sockets, 4

major axis, 148

making a backup, 13

manual, 1

matrix, 81

MDV, 10

mechanical push-buttons, TV, 9

memory, 13

menu, 85

Microdrive, 5, 13, 104

Microdrive filing, 108

Middle C, 160

minor axis, 148

mixing variable types, 41

MOD, 48

MODE, 144

monitor, 2

monitor cable, 3

MOVE, 153

mugtrap, 63

multistatement line, 26

musical scale, 163

names file, 115

nested loops, 57

network connections, 104

NEXT, 55

noise, power supply, 2

number abilities, 43

number functions, 44

number zero, 22

number-guessing game, 64

octave, 161

on/off switch, 5

OPEN, 107

open file, 111

OPEN_IN, 110

OPEN_NEW, 110

OR, 53

out of range message, 79

outer loop, 57

outline plan, 93

OVER, 137

overflow message, 47

PAN, 137

Panda adaptor, 3

PAPER, 142

parallel interface, 171

parameter, 89

parity, 171

pass through loop, 51

PAUSE, 29

PENDOWN, 153

PENUP, 153

piano keyboard, 161

pitch, 159

pitch number, BEEP, 161

pitch numbers list, 162

pixels, 144

placing names into array, 120

POINT, 156

polar co-ordinates, 153

positive integer, 22

POWER plug, 1

power supply, 1

precise dimensioning, 124

precision, 23

precision of numbers, 45

PRINT, 21

print modifiers, 25

PRINT modifiers, 137

procedures, 88

program, 20

program design, 91

program mode, 20

programmed function keys, 12

programming languages, 20

Psion copyright notice, 14

QL colour numbers, 141

quotes, 24

- radian, 149
- random access file, 109
- random walk, 152
- range of angles for ARC, 150
- READ, 42
- reading files, 116
- RECOL, 157
- record, 107
- relative co-ordinates, 149
- REM command, 16
- REPEAT, 50
- repeat commands, 50
- reserved words, 12, 34
- RESET button, 7, 12
- resolution, 144
- RESTORE, 43, 76
- RND, 54
- ROM socket, 6
- rounding numbers, 46
- rows and columns, 26
- running total, 59

- SCALE, 143, 156
- score procedure, 99
- SCR, 133
- screen, 104
- screen co-ordinate map, 132
- screen co-ordinates, 130
- screen display, 7
- SCROLL, 135
- scrolling 25
- SDATE, 82
- SELECT, 85
- selective scroll, 136
- Selmor, 5
- semicolon effect, 26
- semicolons, 24
- semitones, 161
- serial files, 108
- serial interface, 171
- serial printers, 171
- series of arcs, 151
- sharp, 161
- SHIFT keys, 11
- simple melody, 164
- slicing, 70
- socket strip, 5
- sound, 158
- source device, 114
- spacebar, 11

- spaces importance, 21
- SQRT, 45
- standard form, 47
- stave, 160
- STEP, 58
- STR\$ warning, 84
- string, 24, 67
- string functions, 67
- string matrix, 81
- string slicing, 70
- string variable, 34
- STRIP, 139
- structured program, 92
- subroutines, 88
- subscripted variables, 78

- tempo, 165
- terminator, 59
- THEN, 52
- three-pin plug, 1
- title lines, 101
- TO, 71
- totalling numbers from file, 117
- touch pads, TV, 9
- tuning TV, 7
- TURN, 153
- TURNT0, 154
- turtle graphics, 153
- TV cable, 4
- TV receiver, 2
- TV tuning, 7
- TV tuning controls, 8

- UNDER, 139
- underline mark, 14
- underscore sign, 111
- updating file, 126
- upper-case, 11
- use of procedures, 95

- variable conversion, 87
- variable name, 33

- warbling note, 165
- warning, 74
- warning note, 165
- window, 129
- WINDOW, 131
- wrap number, 167

APPLE II	COMMODORE 64	MEMOTECH	SOFTWARE TITENS
APPLE II PROGRAMMER'S HANDBOOK	WARGAMING	MEMOTECH COMPUTING	WORKING WITH dBASE II
0 246 12027 4 £10.95	0 246 12410 5 £6.95	0 246 12408 3 £5.95	0 246 12376 1 £7.95
AQUARIUS	SOFTWARE 64: PRACTICAL PROGRAMS FOR THE COMMODORE 64	THE MEMOTECH GAMES BOOK	USING YOUR MICRO
THE AQUARIUS AND HOW TO GET THE MOST FROM IT	0 246 12266 8 £5.95	0 246 12407 5 £5.95	COMPUTING FOR THE HOBBYIST AND SMALL BUSINESS
0 246 12295 1 £5.95	INTRODUCING COMMODORE 64 MACHINE CODE	ORIC-1	0 246 12023 1 £6.95
ATARI	0 246 12338 9 £7.95	THE ORIC-1 AND HOW TO GET THE MOST FROM IT	DATABASES FOR FUN AND PROFIT
GET MORE FROM THE ATARI	40 EDUCATIONAL GAMES FOR THE COMMODORE 64	0 246 12130 0 £5.95	0 246 12032 0 £5.95
0 246 12149 1 £5.95	0 246 12318 4 £5.95	THE ORIC PROGRAMMER	FIGURING OUT FACTS WITH A MICRO
THE ATARI BOOK OF GAMES	DRAGON	0 246 12157 2 £6.95	0 246 12221 8 £5.95
0 246 12277 3 £5.95	THE DRAGON 32 AND HOW TO MAKE THE MOST OF IT	THE ORIC BOOK OF GAMES	INSIDE YOUR COMPUTER
BBC MICRO	0 246 12114 9 £5.95	0 246 12155 6 £5.95	0 246 12235 8 £4.95
ADVANCED MACHINE CODE TECHNIQUES FOR THE BBC MICRO	THE DRAGON 32 BOOK OF GAMES	T199/4A	SIMPLE INTERFACING PROJECTS
0 246 12227 7 £6.95	0 246 12102 5 £5.95	GET MORE FROM THE T199/4A	0 246 12026 6 £6.95
ADVANCED PROGRAMMING FOR THE BBC MICRO	THE DRAGON PROGRAMMER	0 246 12281 1 £5.95	PROGRAMMING
0 246 12158 0 £5.95	0 246 12133 5 £5.95	VIC-20	COMPLETE GRAPHICS PROGRAMMER
THE BBC MICRO: AN EXPERT GUIDE	DRAGON GRAPHICS AND SOUND	GET MORE FROM THE VIC-20	0 246 12280 3 £6.95
0 246 12014 2 £6.95	0 246 12147 5 £6.95	0 246 12148 3 £5.95	THE COMPLETE PROGRAMMER
BBC MICRO GRAPHICS AND SOUND	INTRODUCING DRAGON MACHINE CODE	THE VIC-20 GAMES BOOK	0 246 12015 0 £5.95
0 246 12156 4 £6.95	0 246 12324 9 £7.95	0 246 12187 4 £5.95	PROGRAMMING WITH GRAPHICS
DISCOVERING BBC MICRO MACHINE CODE	ELECTRON	ZX SPECTRUM	0 246 12021 5 £5.95
0 246 12160 2 £6.95	ADVANCED ELECTRON MACHINE CODE TECHNIQUES	AN EXPERT GUIDE TO THE SPECTRUM	WORD PROCESSING
DISK SYSTEMS FOR THE BBC MICRO	0 246 12403 2 £6.95	0 246 12278 1 £6.95	CHOOSING A WORD PROCESSOR
0 246 12325 7 £7.95	ADVANCED PROGRAMMING FOR THE ELECTRON	INTRODUCING SPECTRUM MACHINE CODE	0 246 12347 8 £7.95
HANDBOOK OF PROCEDURES AND FUNCTIONS FOR THE BBC MICRO	0 246 12402 4 £5.95	0 246 12082 7 £7.95	WORD PROCESSING FOR BEGINNERS
0 246 12415 6 £6.95	ADVENTURE GAMES FOR THE ELECTRON	LEARNING IS FUN: 40 EDUCATIONAL GAMES FOR THE SPECTRUM	0 246 12353 2 £5.95
INTRODUCING THE BBC MICRO	0 246 12417 2 £6.95	0 246 12233 1 £5.95	FOR YOUNGER READERS
0 246 12146 7 £5.95	ELECTRON GRAPHICS AND SOUND	MAKE THE MOST OF YOUR ZX MICRODRIVE	BEGINNERS' MICRO GUIDES: ZX SPECTRUM
LEARNING IS FUN: 40 EDUCATIONAL GAMES FOR THE BBC MICRO	0 246 12411 3 £6.95	0 246 12406 7 £5.95	0 246 12259 5 £2.95
0 246 12317 6 £5.95	ELECTRON MACHINE CODE FOR BEGINNERS	THE SPECTRUM BOOK OF GAMES	BEGINNERS' MICRO GUIDES: BBC MICRO
TAKE OFF WITH THE ELECTRON AND BBC MICRO	0 246 12152 1 £7.95	0 246 12047 9 £5.95	0 246 12260 9 £2.95
0 246 12356 7 £5.95	THE ELECTRON PROGRAMMER	SPECTRUM GRAPHICS AND SOUND	BEGINNERS' MICRO GUIDES: ACORN ELECTRON
21 GAMES FOR THE BBC MICRO	0 246 12340 0 £5.95	0 246 12192 0 £6.95	0 246 12381 8 £2.95
0 246 12103 3 £5.95	HANDBOOK OF PROCEDURES AND FUNCTIONS FOR THE ELECTRON	THE SPECTRUM PROGRAMMER	MICROMATES
PRACTICAL PROGRAMS FOR THE BBC MICRO	0 246 12416 4 £6.95	0 246 12025 8 £5.95	SIMPLE ANIMATION
0 246 12405 9 £6.95	PRACTICAL PROGRAMS FOR THE ELECTRON	THE ZX SPECTRUM AND HOW TO GET THE MOST FROM IT	0 246 12273 0 £1.95
THE COLOUR GENIE	0 246 12362 1 £7.95	0 246 12018 5 £5.95	SIMPLE PICTURES
MASTERING THE COLOUR GENIE	21 GAMES FOR THE ELECTRON	WHICH COMPUTER?	0 246 12269 2 £1.95
0 246 12190 4 £5.95	0 246 12344 3 £5.95	CHOOSING A MICROCOMPUTER	SIMPLE SHAPES
COMMODORE 64	40 EDUCATIONAL GAMES FOR THE ELECTRON	0 246 12029 0 £4.95	0 246 12271 4 £1.95
BUSINESS SYSTEMS ON THE COMMODORE 64	0 246 12404 0 £5.95	LANGUAGES	SIMPLE SOUNDS
0 246 12422 9 £6.95	TAKE OFF WITH THE ELECTRON AND BBC MICRO	COMPUTER LANGUAGES AND THEIR USES	0 246 12270 6 £1.95
ADVENTURE GAMES FOR THE COMMODORE 64	0 246 12356 7 £5.95	0 246 12022 3 £5.95	SIMPLE SPELLING
0 246 12412 1 £6.95	IBM	EXPLORING FORTH	0 246 12272 2 £1.95
COMMODORE 64 COMPUTING	THE IBM PERSONAL COMPUTER	0 246 12188 2 £5.95	SIMPLE SUMS
0 246 12030 4 £5.95	0 246 12151 3 £6.95	INTRODUCING LOGO	0 246 12268 4 £1.95
COMMODORE 64 DISK SYSTEMS AND PRINTERS	LYNX	0 246 12323 0 £5.95	GRANADA GUIDES: COMPUTERS
0 246 12409 1 £6.95	LYNX COMPUTING	INTRODUCING PASCAL	0 246 11895 4 £1.95
THE COMMODORE 64 GAMES BOOK	0 246 12131 9 £6.95	0 246 12322 2 £5.95	
0 246 12258 7 £5.95	COMMODORE 64 GRAPHICS AND SOUND	MACHINE CODE	
0 246 12342 7 £6.95		Z80 MACHINE CODE FOR HUMANS	
		0 246 12031 2 £7.95	
		6502 MACHINE CODE FOR HUMANS	
		0 246 12076 2 £7.95	

The Sinclair QL is by far the most powerful computer ever made available at such a competitive price. With such a sophisticated machine, however, there is so much more to learn! This book has been written to help you get ahead fast. Very many practical programs are provided for you to enjoy as you learn how to get the best from this superb microcomputer – such as elegant and creative programming techniques, how to achieve brilliant graphics and sound, interesting special effects, efficient use of the microdrives, useful database applications, and much more. Also it shows how to avoid some of the problems that may arise. This highly practical book – which is based on the daily use of the machine and not on second-hand information – will put you well on the path to becoming an expert.

The Author

Ian Sinclair is well known as an author and past contributor to journals such as *Personal Computing World*, *Computing Today*, *Electronics and Computing Monthly*, *Hobby Electronics* and *Electronics Today International*. He has written over fifty books on aspects of electronics and computing, mainly aimed at the beginner.

Also from Granada

QL SuperBASIC

A. A. Berk

0 246 12596 9

Front cover illustration by Tony Roberts

GRANADA PUBLISHING

Printed in Great Britain

0 246 12595 0

£5.95 net