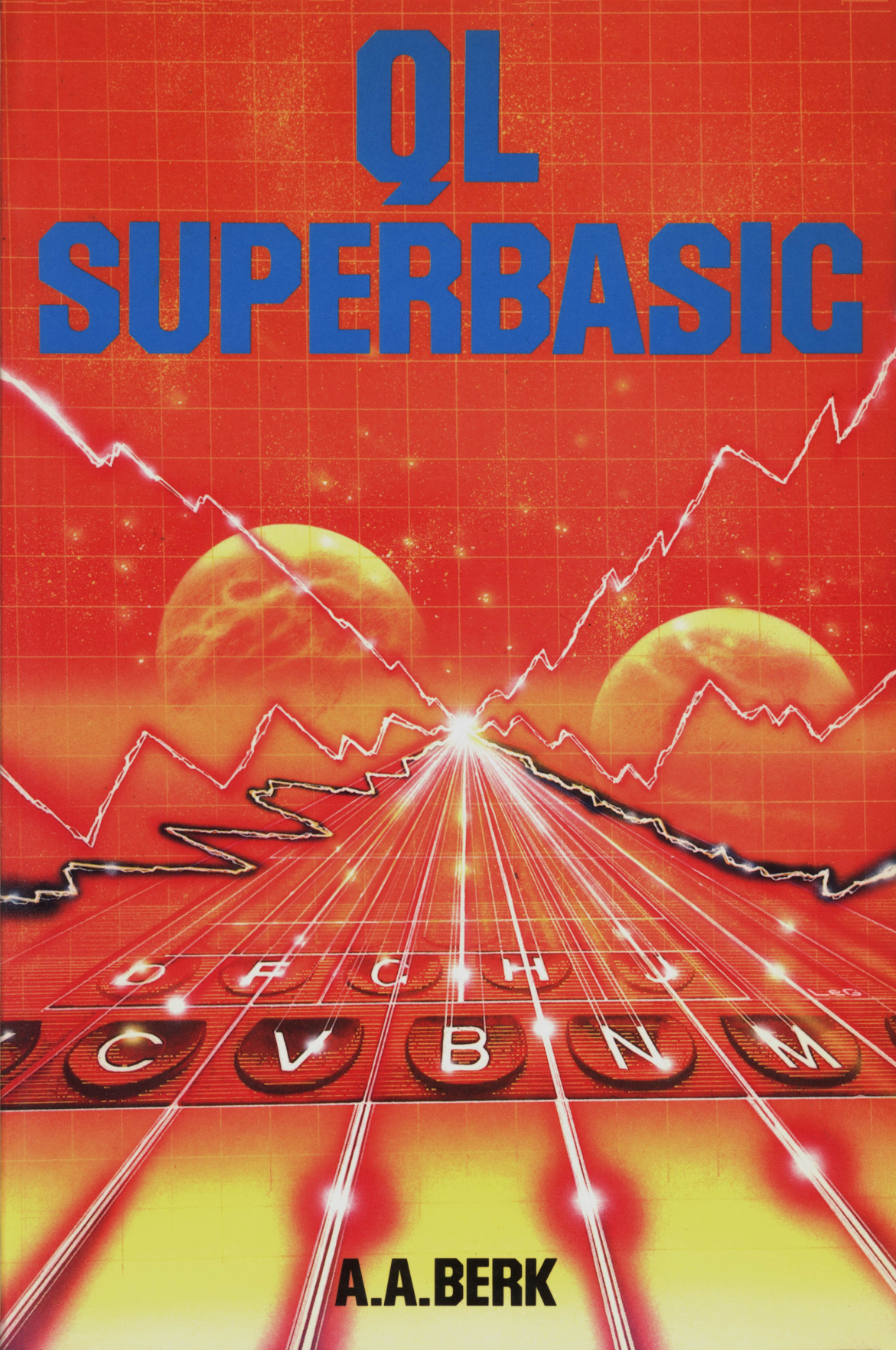
QL SUPERBASIC



A.A.BERK

QL SuperBASIC

Also for QL users

QL Computing
Ian Sinclair
0 246 12595 0

QL Gamesmaster
Kay Ewbank, Mike James and S. M. Gee
0 00 383066 7

QL SuperBASIC

A. A. Berk

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 © A. A. Berk 1984

British Library Cataloguing in Publication Data
Berk, A. A.
QL SuperBASIC.
1. Sinclair QL (Computer)—Programming
2. SuperBASIC (Computer program language)
I. Title
001. 64'24 QA76. 8. S625

ISBN 0-246-12596-9

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

Preface

1	Introduction to the QL	1
2	Introduction to Programming	7
3	Starting with SuperBASIC	27
4	Input/Output	46
5	Loops and Decisions	63
6	Strings and Arrays	91
7	Calculation and Standard Functions	104
8	Graphics and Sound	112
9	Microdrives, Files and Devices	137
10	Procedures, Subroutines and Functions	150
Appendix:	Alphabetic List of Keywords, Functions and Operators (indexed by chapter)	159
Index		163

Preface

The QL is short for 'Quantum Leap'. Its concept marks something of a departure from the norm in the realm of microcomputing. It has similar memory and computing capacity to many medium-sized business machines, but at a cost which makes it affordable to everyone.

The internal microdrives also mean that the storage of data and programs for later use is straightforward, and always accessible. You can store long documents from the word processor, business-type graphics displays, or database-type information for later retrieval and analysis.

This book is concerned, however, with the computer language contained within the machine, and available on power-up. You have a machine capable of accepting program commands in a dialect of one of the most well known programming languages in the world. The dialect is called *SuperBASIC*. It is an enhanced form of BASIC, and again if you are used to using BASIC, you will be pleasantly surprised by the structure and facilities offered.

The book is written at a level which will be suitable for the determined beginner, or the experienced person who needs examples of a tried and tested nature to help him pick up SuperBASIC. The material is also most suitable for later reference, and will be found useful for this purpose even after the language has been learned.

The first version of the QL to appear in the market had a number of software bugs in the operating system and many mistakes in the manuals which come with the machine. It should be borne in mind that *every one of the many examples in this book has been tested and checked on the machine*, and you will find them invaluable in helping to sort out the errors in the manuals, while giving you a feel for the use of SuperBASIC.

Dr A. A. Berk

Chapter One
Introduction to the QL

This chapter assumes that you have followed the QL manual's instructions on plugging your computer in, and now have a display and keyboard in front of you to work with. We shall look at some of the simple operations you must be able to perform with these devices to understand the rest of the book.

The computer

The electronics of the QL computer occupy a large electronic circuit board which is fitted neatly beneath the keyboard, within the black plastic case. This case also houses the two microdrive units, to the right. There are several opening hatches and sockets in the back and sides of the case through which access is made to the electronic circuit board within. This allows a large variety of extras to be fitted to the main unit, from printers to increased memory. None of these need concern us here, as we shall not be looking at the electronics in any way.

The main characteristic of the machine, as far as we are concerned, is its ability to communicate with us through keyboard and screen. Via these two units, we shall learn to command it to perform any task of which it is capable, by writing a set of instructions called a *program*.

If you wish to add a printer to the machine, you should buy a printer which is stated to be capable of plugging into the QL, and ensure that you have the correct cables for the job.

SuperBASIC

To communicate with anything, be it man or machine, you must first discover the basic words it is capable of understanding. This is called its *language*, and your first task is to learn the QL's language, which is called SuperBASIC.

The basic elements of SuperBASIC are called *statements*. Each statement provides a single command to the computer. For instance, there are

statements to write something on the screen, add two numbers together, beep the internal beeper, and so on. A program is simply a set of such statements, written using the special English-like words of the language.

Using the QL

It is assumed that you have unpacked your machine, plugged it together, and are now sitting in front of the screen ready to go.

Before going any further, look round to the right end of the keyboard, and find the *reset* button poking out from the side of the machine. Press this a few times to see the effect on the machine. This button is used to take you out of situations where either something has gone wrong with the machine and it will not respond any more, or you wish to start again completely from scratch. When you press reset, any program which may be in the machine is lost. If it is on a microdrive, you can always recall it, but if not it is gone forever. Do not make a habit of pressing reset, as it is a rather drastic manoeuvre!

Each time you press the reset button, the screen will change to a random pattern of coloured dots for about a second. This illustrates the complete set of dots which can be separately displayed in a number of colours by a computer program. The display then clears to show two coloured boxes, one of which is a pair of alternatives for you to choose, and the other a copyright message. The alternatives concern two of the *function keys* on the left-hand side of the keyboard. F1 and F2 are the top two of five function keys labelled F1 to F5. These 5 keys are used for special purposes in programs, but when first switched on, two of them are used to inform the QL whether you are using a normal TV, or a special colour monitor. A colour monitor will give considerably better output than a TV, but they are expensive. A TV is certainly adequate for experimentation and program writing.

If you have a TV, pressing F2 will ensure that the QL's output picture does not disappear off the edge of the screen. However, some sacrifice is made to the total amount of screen area which is otherwise available on the machine.

Whether you have a TV or a monitor, press reset, and then press F1 and watch the little dark red square in front of the left-hand microdrive slot. If you missed it, press reset and try again. You should see the square light up and hear a slight whirring noise as the microdrive spins for a short time.

The keyboard

Following the instructions of the last paragraph will have changed the screen to show three windows, white, red and black. These three windows are used for the various types of output which your QL will want to

communicate to you. For instance, try pressing any letter, and keep it pressed for two or three seconds. After a short time, the screen shows a line of letters repeating over and over again. This is in the black window at the bottom of the screen. The QL is communicating the fact that it has recognised a keyboard key being pressed, by displaying the letter indicated on the key top. The bottom window is called the *command window*; you will see your commands displayed there as you type them in.

The other two windows are used for displaying output from your programs, and the actual program itself. We will see this later. For the moment, let us use the command window.

You should have a line of letters on the screen, ending with a flashing red blob. This blob is called a *cursor*, and it always shows you where your next input will appear in the command window. Try pressing another key while watching the cursor. If you keep the key pressed for too long, the character which appears on the screen will start to repeat quite fast. You will have to become used to pressing keys for just a short time, or all your input to the machine will cause errors. Try typing in your name and address carefully, preventing any doubling of letters. Do not worry about any typing errors, you will learn how to correct them later – just keep typing. You will have to use the long SPACE bar at the bottom of the keyboard to print spaces between your words. If you continue typing letters, the cursor will eventually fall off the edge of the screen. In fact it has gone back to the far left-hand side of the screen, but you will not be able to see it until you have typed enough letters to bring it back into view. Try this.

If you are using a TV, the screen picture will not be very clear, and we will now go back to the normal TV display. However, you should bear in mind that there are three windows on display here, as this is important for later.

Now press the reset button, and answer with F2 whether you have a TV or a monitor. The flashing cursor will now be visible again, but this time it is magenta in colour. Try pressing a key for a second or two again, and you will see larger easy-to-read letters displayed in the command window. Try typing your name and address again, to pick up the feel of the keyboard. As you continue typing, the line will eventually become too long to fit on the screen, and the cursor will jump back to the left of the screen. Everything typed in up to that point will move or *scroll* up the screen. Keep typing, and long lines will be spread over several screen lines as you type. The black command window can only show four lines of text, and if you type so far that scrolling actually takes earlier text off the window at the top, that text is lost irretrievably.

The letters which you have produced will have been displayed up to now in lower-case. To produce capitals, press one of the SHIFT keys, and press any letter while keeping the SHIFT key down. If you let the SHIFT key back up again, subsequent letters will be in lower-case. Practise writing your name and address again, but this time with the correct capitals, if you have not already done so. Again, you should be concentrating on gaining a feel

for the keyboard and display as you go. You will find that there is a SHIFT key at each end of the keyboard, for convenience.

You will notice that some of the keyboard keys have two characters on their tops. The upper characters are always obtained using SHIFT as for capital letters. You should now take a few minutes to go through each character on the keyboard, and try each with and without SHIFT to see the effect. This will also show you the complete character set which you can use on the QL.

For programs in general, it is usual to input the commands and statements using upper-case letters. It would be a nuisance to have to keep the SHIFT key down the whole time, especially as this also shifts the number keys at the top of the keyboard to other less frequently used characters. It is usual for a computer keyboard to have a CAPS LOCK key which changes the keyboard into upper-case letters, but leaves the numbers and other keys the same. Press CAPS LOCK just once, and try pressing the keyboard keys. You will see that numbers and other non-letter keys are as if in the lower-case mode, but all letters are now capitals. This mode remains in effect until either the computer is reset, or CAPS LOCK is pressed again. Each time CAPS LOCK is pressed, the mode changes.

You will find that some of the keys do not produce an output. For instance, all the function keys, and the ESC character do not print anything. These are used for other functions. You will find that the same is true of CTRL and ALT. ALT is used by the Psion programs which come with your machine, and we shall see how CTRL is used shortly. You will also have tried the keys with arrows on top by now, and the ENTER keys. They all affect the display, but do not necessarily print any thing up. However, when you press ENTER, you will probably find that *bad line* has been displayed. This tells you that the computer is checking on what you are typing, and is not impressed! It is looking for a valid command, and all it sees is a meaningless jumble of letters.

Editing

Unless you are lucky, you will probably find that you have made at least one typing error so far. It is important to be able to edit out your typing mistakes, and replace them with the correct character.

Try typing a letter, and then press the CTRL key. While keeping it down, press the *left-arrow* key – i.e. the key with an arrow on it which is pointing to the left. The letter you just typed in will disappear, and the cursor will have moved to the left. If you keep the CTRL and left arrow pressed, the cursor continues stepping to the left, erasing everything as it goes. It will eventually stop at the start of the line you were typing, and wait for your next move. This method of erasing allows you to alter your work.

Input a few letters, and press the left-arrow key again, but this time do not

press CTRL. This will move the cursor back along the line, and place it over one of the letters you have just typed. You can still see the letter, as the cursor is transparent. Again, if you keep the arrow key pressed for too long, it repeats, and the cursor moves quite fast along the line. To move it back along the line the other way, press the right-arrow key. By this means, you can position the cursor over any letter on the line. Notice that the cursor will not move past the beginning or end of the line of letters.

Now position the cursor over any letter, and press the right-arrow key *while holding down CTRL.* This will cause the letter under the cursor to disappear, and the letters to the right to close up to take its space. Thus, you can erase the letter at the current cursor position, or the one before it, using one or other of these arrow keys with CTRL. Try typing any letter, without the CTRL key pressed. You will see that letters are inserted into the line at the cursor position, and text to the right of the cursor moves aside to allow the insertions in. By this means, you can change any letter in the line. Simply position the cursor over the letter to be changed, delete it using CTRL with right-arrow, and type in the new letter. Try this now.

You should now be able to have a go at typing a long sentence accurately and using the editing features above as required. If you finish by pressing the ENTER key, the sentence is entered into the computer for checking, and the *bad line* error will be displayed again on the screen.

A command

To see how the command window should be used, we will practise using the BEEP command. This simply produces a single continuous tone from the internal loudspeaker within the case.

To input a command to the computer, you must first of all have a clear line in the command window, with the cursor over to the left of the screen. The quickest way to achieve this is to press ENTER. Do not worry about any error message.

The computer is now waiting for a command. Type in the following exactly as shown:

```
BEEP 3000,100
```

This consists of four letters, followed by a space, then four digits, a comma and three more digits. The cursor should now be just after the last 0 typed. If you made a typing error, edit the line until it looks exactly like the one above. This is a valid command, but the computer will not accept it until you press ENTER. This is a general rule, as you are given the chance to edit your typed line, until correct, before the computer checks and follows the command. Now press ENTER, and you will see, or rather hear, the result. The cursor will also have jumped back to the beginning of the line, and the characters of the command will have been scrolled up by one line – unless

you held your finger on ENTER for too long, in which case the command has probably disappeared. Do not worry about this: once a command has been used in this way, it cannot be reused without retyping it.

If you make a mistake, and do not correct it before pressing ENTER, you will be told so by the computer. Try typing in the above, but miss out the 100 at the end. The computer tells you that the BEEP command has not been used correctly. The first number in the command above tells the computer the duration of the beep, and the second number tells it which note to play. If one of these *parameters* is missing, the computer produces an error message.

Before you experiment further with the BEEP command, you should know how to stop it if necessary. If you type in the following, you will see what this means:

```
BEEP 0,100
```

The beep is still going, and will do so until you reset or type in BEEP on its own followed by pressing ENTER. Try stopping the beep by one of these methods, and then try changing the numbers in the command. The formal useful ranges for these numbers is from 0 to 32767 for the first number, and from 0 to 255 for the second.

Summary

- When the reset button is pressed, everything in memory is lost, and the computer starts again from the beginning.
- The keyboard is in lower-case mode at the beginning, and upper-case letters can be produced by pressing SHIFT while letters are typed.
- CAPS LOCK will cause the keyboard to go into upper-case mode whereby letters are only available in upper-case. Other keys are unaffected.
- CTRL with left-arrow deletes the character just before the cursor, while CTRL with right-arrow deletes the character under the cursor.
- Letters are automatically inserted in the line to the right of the cursor, and the rest of the text shifts over accordingly.
- ENTER causes the computer to consider the line of characters just typed as a command; it is checked for validity, and an error message displayed if incorrect.

Chapter Two
Introduction to Programming

If you are a beginner to programming, or if you are rather rusty and have not used a computer for some time, you will find this chapter very useful as a start or restart to the subject. The chapter does contain some important information on SuperBASIC, and even if you are experienced, you should still read through the summary at the end of this chapter before moving on to the next chapter.

This book is concerned with one particular aspect of the QL. It is aimed at describing the *software*, or programming of the machine. The QL comes with very sophisticated packages of ready-made software, which have been developed on a large computer, and stored on microdrive cartridges for you to use as you wish. However, these packages are aimed at one or two aspects of the use of the computer, and you can only gain the full power of the machine by learning to produce computer programs yourself. Progams are simply lists of instructions written using words which the computer can understand. As long as you know these words, and their precise definition, you can make the computer perform any task which you can imagine, and many more. There are a few simple rules to learn, and the power of the computer is entirely at your disposal.

To make a computer do precisely what you want, you must first learn to believe in and fully apply the First Law of Computing - namely that *computers are stupid*. They are only capable of performing a set of tasks blindly and logically. This universal fact applies to all computers, no matter how complex and apparently formidable. If you can convince yourself firmly and completely of this simple attribute of all computers, progamming will come to you in a natural and intuitive manner. If you imagine for a moment that the computer is intelligent, you will forget that you have to tell it to do every single thing that it will ever be capable of, and this stupid machine will get the better of you!

To be more technical about the central fact presented above, you must analyse and fully describe any task in a simple and logical manner, and hence break the task down into elements, or *statements* using words which the computer can understand and follow. Many of the mistakes or *bugs* which are found with computers can be traced to a lack of application of this first law of computing.

This chapter is largely involved with helping you to think in small logical steps. The size and character of each step depends on the language which the computer is able to 'speak'. The QL uses a dialect of a famous *high level language* called *BASIC* which stands for *Beginners All-purpose Symbolic Instruction Code*. This acronym, however, should not be taken as implying that BASIC is somehow inferior, or only of use to beginners. BASIC is used to write many types of serious software, and is by no means confined to simple applications. You will come to appreciate how broad are the tasks that BASIC is capable of as you work through this book.

The machine

To see how BASIC fits into the machine, it is worth knowing a little about the computer. For instance, all microcomputers use a silicon chip called an *MPU* which stands for *microprocessor unit*. This is the heart of the machine, and its most important attribute is simply understanding and executing instructions. However, the MPU's instructions are written in a language called *machine code*, which is not very human-orientated. We say that machine code is a *low level language*. In order to make the MPU execute near-English instructions, such as those used by BASIC, a translation has to occur from BASIC to machine code. This translation is performed continuously by the machine, and you do not notice it happening. We say that the translation occurs in a manner which is *transparent* to the user. Machine Code will not be described here, but you should remember that it is theoretically possible to produce a translation from any of the many high level languages which are in existence down to machine code. BASIC is simply one of these languages, and is currently the most popular, as it is one of the simplest to learn.

Let us begin by learning to break tasks into smaller units which can be programmed as BASIC statements.

Logical thinking

As you learn to program you will probably find your thought processes becoming more and more orientated towards thinking precisely and analytically. One of the best ways to begin this process is to try to imagine how you would explain a simple activity to a rather stupid but highly logical child. A good example is the task of getting to work or school in the morning. One of the reasons why this is a good example is that you have probably never analysed the actions you take to achieve the end. For instance, where would you start to explain this task? Would you say that you get up, have breakfast, say goodbye to the family and then take a train or bus? The logical child might imagine you walking around all day in your

pyjamas! He might think that you perform this task while asleep since you have not explained that you wake up first.

Further, you would have difficulty explaining to him how you know when to wake up when you are asleep, and so on. As you can see, analysing an apparently simple task can be taken to an ultimate and cumbersome limit, and it all depends upon how much the child understands. This is exactly the case with computing. You have to know exactly how far you have to go in explaining the action. In short, you have to know what words the computer understands.

An example of a problem

Let us take the simple example of sorting three numbers into ascending order. The numbers are 9,1 and 3. You can do this without thinking, but how would you explain the actions you have taken to achieve the end?

You might assume that the computer understands *relations* such as = and 'greater than' and 'less than'. Also, that it understands the words 'swop' and 'if'. You could thus tell it to SWOP the first two numbers IF the first is GREATER THAN the second. This would give 1,9,3. The same treatment could then be applied to the second two numbers, and this would do it in this particular case. How many other concepts have you used to write down this sequence of events?

You have probably used at least two central ideas without thinking about them. These are *storage* and *labelling*. We have labelled the numbers 'first', 'second' and 'third', and we have assumed that they are stored somewhere in the computer other than simply on the paper in front of your eyes.

Another two fundamental ideas in computing are those of the *input* and *output* of quantities being worked on. We have assumed, in the example above, that the computer has a means of inputting the numbers, and outputting or displaying the answer to each step of the sequence.

As you can see, there are some important concepts to sort out before we can even begin to converse with the computer.

Labelling and storage

Whenever we deal with a quantity, or *variable*, in any way, the first question should always be *how is it labelled and stored?* We cannot even talk to each other about an object until we have a name for it, and computers are the same. Once a quantity is labelled, it can be manipulated, changed and generally used within the computer.

Another important activity is that of creating space within the computer for the storage of a variable. The computer does this automatically. Whenever we label something, it immediately reserves a specially labelled

storage location within its memory for that variable. The term *variable* can include a large variety of different things from 10-figure numbers to letters and characters such as full-stops and commas. All the characters shown on the keyboard of the QL can be used within a variable, and printed out on the screen, or used for other purposes in a computer program.

When the computer sets aside some storage space for a variable, it is important that it reserves the right kind of storage space, and the right amount. To do this it has to know whether the variable is meant to store a number or a collection of general characters. In BASIC, the actual make-up of the variable's name tells the computer what kind of variable it is.

Identifiers

Most of the concepts we shall meet in computer programming have special technical words which are used to describe them. For instance, the word *quantity* above has been replaced by *variable*. Similarly, the word *label* is usually replaced by *identifier*, or sometimes *name*. We say, therefore, that we give an identifier to a variable, meaning we name or label the quantity.

All computers have rules about the make-up of an identifier for a variable, and the QL is no exception. These rules help to prevent the computer becoming confused. For instance, if you were allowed to call a variable by the name 12.4, how would the computer know whether 12.4 should be treated as a number or a name?

The QL allows a sequence of up to 255 letters, numbers and underscore symbols (_), to be a variable identifier. However, the identifier has to start with a letter, to save confusion. For instance, you can use N or Number or NUMBER_3 etc., for an identifier. You cannot use _number, or 212Number because the first characters of these identifiers are not letters. Also, the QL does not distinguish between upper- and lower-case letters for identifiers, and the following names are all regarded as the same by the QL:

Cost cOST CoSt

As a general rule, it is worth trying to stick to identifiers which mean something to you in the program. It helps both with writing and with understanding the program. For instance, suppose you were trying to compute the price of an article, given raw material cost, labour cost, overheads cost and profit. It would be difficult for you to remember the meanings of the variables if you called them A, B, C, D and so on. It is much better to use: PRICE, RAWMAT, LABCST, OVRCST and PROFIT. These identifiers are self-explanatory. Notice how the quantities have been abbreviated. It is usually worth keeping the names to a minimum number of letters, purely for convenience of typing.

Storage

When an identifier is used, it is important that the computer reserves the right type of memory storage for that variable, as a variable can be set equal either to a number, or a collection of letters and other characters called a *string*. There are, thus, two types of variable, *numeric* and *string*.

The computer distinguishes between the two types of variable by the last character of the identifier for that variable. If the variable is to be a string of general characters, then a $ sign must be used as the last character of the identifier. This identifies the name as being that of a string variable, and the computer reserves the appropriate type of memory storage for it. For instance, Name$ could contain a string of letters, and the computer automatically reserves space for this type of data in its memory. The absence of a $ implies that the variable is numeric, and the computer will reserve memory space for a number.

There is another suffix which may be used too; this is the % sign. This implies that the variable is an *integer* or whole number, and will be described later.

Assignment

You will have to type in some program statements, make sure that either you have just turned the computer on, or you have pressed the RESET button on the side of the machine before starting this section. To ensure that you do not change the character of the display in an unexpected manner, stick carefully to the examples given. When you have tried a few of them, you can experiment for yourself.

The next problem is to store some data at the location labelled by our identifier. This process also has a technical name; it is called *assignment*, because it assigns a 'value' to a variable. As explained above, this value does not have to be a number in computing. For instance, a variable may have your name and address stored in it. Your name and address is then referred to as the *value* or sometimes the *contents* of that variable.

The word used by BASIC for an assignment statement is LET, and an equals sign. Thus, for instance,

```
LET A = 33.4
```

would cause the computer to reserve some memory for the variable A, and then store the number 33.4 in that memory. To call up this number again, you simply use the name A, as we shall see. This is a numeric assignment, and string assignments are similar, but require some further symbols. The $ symbol has already been introduced. Before continuing, type in the above statement using the keyboard, and finish off by pressing ENTER. You do not need to use capitals, but it is a general practice to use them in computing,

and we shall do so for this book. To save having to press the SHIFT key all the time, press the CAPS LOCK key at the start, and this will ensure that capitals are always printed. LET is a special BASIC word, and is called a *keyword.* All keywords must be followed by a SPACE, as shown, to save confusion as we shall see. The other spaces are included to make the display a little neater, and you should use them if you can.

If you wish to store the name Fred in memory for later use, it is important that the computer does not confuse the word Fred with an identifier. To prevent this, the name Fred has to be surrounded by quotation marks. You can use either single or double quotes in QL SuperBASIC, though double quotes are normally all that is allowed by other machines. The following are valid SuperBASIC statements:

```
LET Name$ = "Fred"
LET NAME$ = 'Fred'
```

and they both set the same variable equal to the string of letters Fred. Again, type these statements into the QL ending with ENTER. Press CAPS LOCK again to select lower-case letters where necessary. Remember the space after the keyword LET.

Remember, also, that the QL sees upper- and lower-case as the same for *identifiers.* Thus Name$ and NAME$ are considered to be the same identifier by the QL. However, this is not the case for the actual data which is stored in that variable. The contents of the variable are stored literally. Thus "Fred" and "fred" are different as contents of a variable, because they use different characters for the first letter. The quotes used as separators around strings are called *literals*, as everything within the quotes is copied literally into memory.

As a final note on assignments, SuperBASIC allows you to omit the LET, if you wish. This is a common abbreviation in BASIC, and saves time. Thus, the statement:

```
LET NAME$ = "FRED"
```

is the same as:

```
NAME$ = "FRED"
```

As there is no keyword in the second statement, the spaces are not necessary, but are included for neatness. It is crucial that you understand that this is not a statement telling you that NAME$ and "FRED" are the same. It is a *command* to the computer to take the letters between the literals, and store them at the memory locations identified by NAME$. The equals sign here is not an ordinary statement of fact, it is an order to the computer to assign the letters of FRED to the variable NAME$, by storing at the memory location labelled by NAME$.

If you remember the problem of rearranging three numbers in ascending order, we had to identify the three numbers by using rather woolly

expressions such as first number, second and third. We can now do much better using the rules and ideas of this section. They could be called A, B and C, for instance, and assignment statements used to set these variables equal to the numbers. We shall see how to do this shortly.

Input and output

The last section has solved the important problem of allowing us to call variable quantities by names which identify them precisely to the computer. The next task is to see how to input and output quantities which are to be worked on.

There are many methods of input to the computer using SuperBASIC, and they are described in the following chapters. For the moment, the last section has identified one method. The numbers can simply be assigned to variables using the LET or = statements within the program itself.

Output requires another BASIC word, PRINT, and allows the contents of a variable to be displayed on the screen.

As explained in Chapter 1, the screen is divided into three main areas. These are called *windows*, and as you type on the keyboard, the characters appear in a window at the bottom of the screen, and you can edit them as you wish. If you are using a TV, then you might be wondering where one of the windows has gone, as there are only two left on the screen. The answer is that two of the windows are overlaid on each other, as each takes up the whole screen. We shall see this shortly. If you are using a video monitor, then the left-hand section of the screen is coloured white, and the other section is coloured red, on switch-on. The red section is used by the program to display its output, and this is where the PRINT statement can be used to display the contents of any variable.

As with most of the English words in BASIC, PRINT is simple to use and understand. You can almost guess how to use it. If you want to display the contents of a variable such as NAME$, you simply tell the computer to PRINT it. Make sure that you have typed in the assignment statement for NAME$ above, and then type in the following ending with the ENTER key:

```
PRINT NAME$
```

Again, PRINT is a keyword, and must be followed by a space. Try leaving it out, and you will see that the computer is doing some checking in the background all the time, and will tell you that there is an error here somewhere. If you see this error (bad name) always check for spaces after the keywords. There is a list of error messages in the QL manual, and when you see one you can use the list to help you find the mistake.

The computer does a tremendous amount of background work to execute the straightforward PRINT command above, but this is of no matter to the programmer. When he types in PRINT NAME$, he simply expects to see

the contents of NAME$ appear on the screen. If you are using a video monitor, then typing in the above will cause FRED to appear in the output window at the right-hand side of the screen. If you are using a TV, however, the output will appear at the top left-hand side of the screen, and will have a red background. This tells you that it is being displayed in the output window on the screen.

A small program

Even though we have only learned two BASIC words, we can write a small program which uses all of the ideas introduced so far. These include breaking the problem into small parts which the computer can understand, identifiers, assignments, input and output. To do this, type in the following:

```
LET A$ = "This is the QL!"      (then press the ENTER key)
PRINT A$                        (then press the ENTER key)
```

If you type the characters above exactly as shown, and press the ENTER key after each line, the result will be an immediate execution of these statements. If you forget the space after the PRINT, a bad line error will occur. However, if you leave the space out after the LET, no error is given, but the display will show an asterisk instead of the contents of A$. This is your first encounter with the really *blind logicality* of the computer, and points out how important it is for you to follow the rules and tell the computer precisely what to do at any time. If you leave the space out after the LET, the computer assumes that you meant to join all the letters up, and thinks that you meant to store the words in quotes in a variable called LETA$, which is a perfectly valid identifier! When you turn the computer on, all string variables, including LETA$ and A$, are empty of data, and this is signified by the asterisk. Thus when the PRINT statement comes to be executed above, it prints the empty string in A$.

The spaces after keywords are only actually necessary if there is a letter, number or underscore character in the next item on the line. However, it is not a bad idea to stick to using spaces after keywords, unless you are cramped for space.

The first statement in the program above is an assignment, which stores the characters between the quotes in a variable called A$. When you press the ENTER key at the end of this line, the assignment is executed then and there by the computer. The second statement simply tells the computer to output the contents of the variable to the output channel, and the words will be displayed there on the screen. The output will appear underneath the last output which was displayed, as the computer carefully avoids overwriting. Again, the red background tells you it is being displayed in the output window.

This simple computer programs shows a number of different concepts.

First, statements can be executed immediately after the ENTER key is pressed. No statement will be executed until the ENTER key is pressed, and it is a familiar sight to see a complete beginner sitting in front of the screen wondering why the computer is not doing anything, if the correct line has been typed in, and ENTER has not been pressed! The ENTER key simpy tells the computer that you are satisfied with what you have written, and you wish it to be executed. Before you press ENTER, you can correct any typing errors as explained in Chapter 1.

Because BASIC statements are so close to English, you will find that you naturally pick up their meaning. However, you should read the definition of each word with great care, since English is not a very precise language – it suffers from the problem that a statement may have a number of interpretations. This is not, however, the case with any computer language. You can write completely clear and unequivocal statements in BASIC, and if you know the rules, the outcome of any set of statements is unique and cannot be argued over. One problem is that large programs containing many hundreds of BASIC words can be too complex to follow easily, and the result may need many hours of work to decipher. You will find that this is even true of programs which you have written yourself, but not looked at for a few days. It is not easy to pick up a computer program and understand how it works. One way of avoiding the confusion is to write the problem as a visual graph of statements before writing the program. Such graphs are called *flowcharts*, and most people use them, or something similar, quite naturally in their everyday life.

Flowcharts

Now that we can talk about objects using labels, give them values, and even print out these values at will, we are ready to return to the first law of computing. Any problem must be broken down into small pieces, each of which is written using words which the computer can understand, and thus the computer is told exactly what to do at any time. However, to write the solution to a problem as a computer program, you must learn a large amount about computer programming, including the meaning of all the words which the language offers.

To save all this learning work before we can even start to understand programming, we use a visual method of breaking problems down and illustrating their action by using a flowchart. A program represented by a flowchart requires no knowledge of the words available in a particular language, and should be able to be translated into any language. However, we will use as many BASIC concepts as possible in the flowcharts which follow to ensure that they are easy to convert into BASIC at a later date.

To show the principles of flowcharting, a very simple flowchart is illustrated in Fig. 2.1. This flowchart allows you to see the various actions

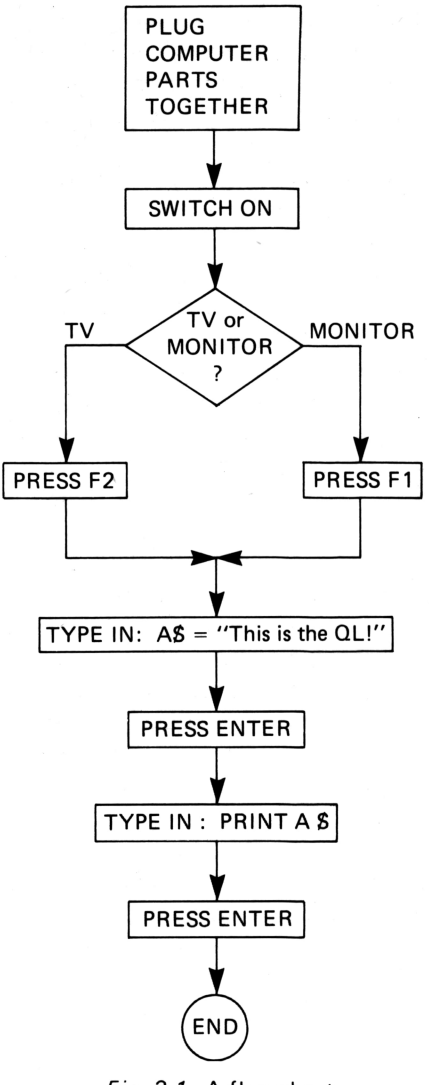


Fig. 2.1. A flowchart.

you would have to take to type in the small program above. It goes into the details of all the actions associated with the computer, and you simply follow the flow by the arrows. It starts by telling you to plug in the computer and the screen. The next part of the chart tells you to switch on. These are all statements in a program, but not statements meant for the QL; they are meant for the human operator to execute. The principle is the same, however. You understand the English statements used, and translate them into actions which you can execute. The next statement, enclosed in a diamond-shaped box, is a decision. It asks a question and you follow the output line from this box depending upon the type of screen you are using. If

you are using a TV, you should know by now that you have to press the key labelled F2 on the keyboard to move to the next stage. If you are using a video monitor, F1 will bring up the correct layout. This ability which humans have for making decisions is reflected in BASIC by a number of different words, and these will be introduced in later chapters. After pressing the correct F key, you are ready to type the program statements. This is performed exactly as above, until you reach the END box. No action is to be taken at this final box, it simply shows you where the chart ends.

This flowchart introduces most of the concepts you need to know about flowcharting: there are several ordinary command or *statement boxes*, there is a *decision box*, and there is an *end box*. In this case, the chart flows from top to bottom of the paper. This is not necessary, and as long as you remember to include arrows, you can arrange your chart in any way you wish.

A flowchart gives a simple, clear method of reading even the most complex of routines, and it is thoroughly recommended as a means of collecting your ideas together for a program. Sadly, as one becomes more proficient in programming, the chore of producing a flowchart is often neglected, and you will rarely see a full flowchart for someone else's program. The resultant confusion which this causes can be overcome by correctly documenting the program while it is being written. Using meaningful identifiers is a great help, but BASIC also allows you to include explanatory remarks within the program, as we shall see. For the moment, remember that a program on its own is difficult to read, and should have some means of explanation either built in, or expressed as a flowchart.

The flowchart of Fig. 2.1 contains a large quantity of information about the way in which you use the QL, and as this would be repetitive if included in every flowchart, it is normally omitted, and just the program statements are included. The flowchart will usually be written as shown in Fig. 2.2.

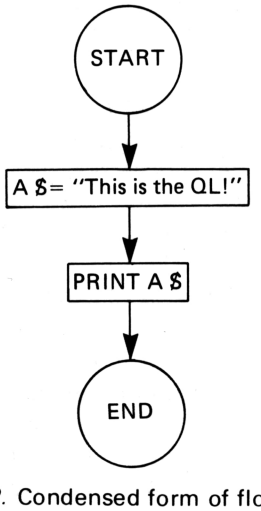


Fig. 2.2. Condensed form of flowchart.

An important concept in computing is that of the *loop*. This is best explained using a flowchart, and will be described next.

Loops

One of the most useful points about a computer system is that it is capable of performing repetitive actions accurately, and without becoming bored. For instance, you might have stored a list of names in a set of variables, and you discover, subsequently, that you have been spelling one of the names incorrectly. Suppose that you would like to change the spelling of the incorrect name. If there are only 5 variables in the list, you could PRINT each of them using 5 successive PRINT statements. This would display the names on the screen, and show up the one with the incorrect spelling. An assignment statement could then be used to correct the name.

As the printing in this operation would have to be performed over and over again, you can think of this as going around in a circle, and we say that you are performing a *loop* of operations. The origin of this word will become clearer when you see a flowchart for the activity shortly. The whole operation would only take you about two minutes, and is hardly a routine which needs to be computerised.

However, suppose that a program has produced a total of 1000 names, each stored in a separate variable, and furthermore the incorrectly spelled name occurs in several of the variables, and you do not even know how many! This is certainly a computer problem, and you would become bored, and be likely to make a number of errors while performing the repetitive task of printing and correcting these yourself. We will see shortly how to write a simple computer program to solve the problem.

Naming the variables

If you remember one of the first concepts introduced above, you will know that we have to decide how the stored names are labelled or identified before we can perform any computer activities on them. Remember that as these names are not numbers, each is to be treated as a collection of characters, and hence is a string. This means that any identifiers we use for them must end in a $ sign.

There are plenty of valid methods of identifying these names. We could use the string identifiers:

A$, B$, C$, etc.

except that we will run out of letters after 26 of them. We could resort to:

AA$, AB$, AC$ ... and so on.

However, this only gives 676 different variables, which is still not enough. Combinations of three letters would be required to give each of the 100 names a separate identifier.

There are other problems associated with this type of identification, and in general you will see that it is essential to use numbers to identify the names wherever you have a large set of objects to store. We naturally use numbers ourselves in normal speech. For instance, when there were three numbers to arrange in ascending order above, it was quite natural to call them first, second and third. It would certainly not have been natural to call them dog, cat and horse! This is partly because we have special associations with this type of labelling, and partly because we want to set the values of these variables in sequential order, and numeric labels are naturally in this type of order. That is, 1 comes before 2 comes before 3, and so on.

Operating upon a large set of names is done by attaching numbers to them, and most computer languages will allow this. In normal usage, we might call the names by the following type of label:

A0$, A1$, A2$, A3$, A4$, ... , A998$, A999$

This labels the names by a numbering system, and identifies them by labels which are quite valid in BASIC – they start with a letter, only use letters and numbers, and end in a $ sign. We can draw a flowchart quite easily for the problem using these identifiers. The numbers are called *indices*, or *subscripts*, and you should regard them as a method of labelling the identifiers numerically. However, to be able to use the numeric part of these labels arithmetically in BASIC, it turns out that we have to enclose the subscripts within brackets. The variables are written as follows:

A$(0), A$(1), A$(2), ... , A$(998), A$(999)

As you can see, even a simple object such as an identifier has to be written according to a precise set of rules. Each identifier must start with a letter, here A. By looking at these labels, you can tell that they are strings by the $ sign, and that they are subscripted. The $ has to come before the subscript. These are rules which you must simply learn and use.

You will see that we will be able to actually operate on the values of the subscripts themselves. This type of variable, having a subscript, is called an *array variable* because it forms an array of pieces of data. These will be introduced more fully in a later chapter.

The problem's flowchart

To continue with the problem, we will further assume that the name which has been misspelled is ANN, and this is to be corrected to ANNE in every variable in which it occurs.

Figure 2.3 shows a flowchart which will solve the problem eventually. The

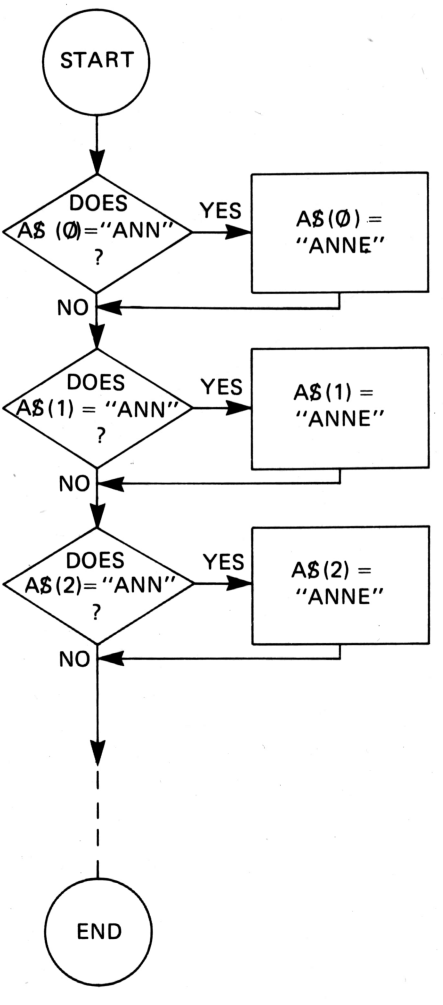


Fig. 2.3. Spelling correction problem.

operations are repetitive, and the first statement asks whether the first member of the set of variables contains ANN. If it does, the assignment statement is executed, and A$(0) is set equal to ANNE. This would have performed a correction to the spelling. Notice how the equals sign is used to ask a question in the decision statement, but in the command statement it is used as a method of telling the computer to set a variable equal to a string. These two different uses of the = will become clearer as we proceed.

The next operation in the chart is the same as the first, but acts on the second variable in the array of names, and so on. The dotted line at the bottom of the chart implies that the process continues and continues until complete. There would be 1000 decision statements, and 1000 assignment statements in the chart if written out in full.

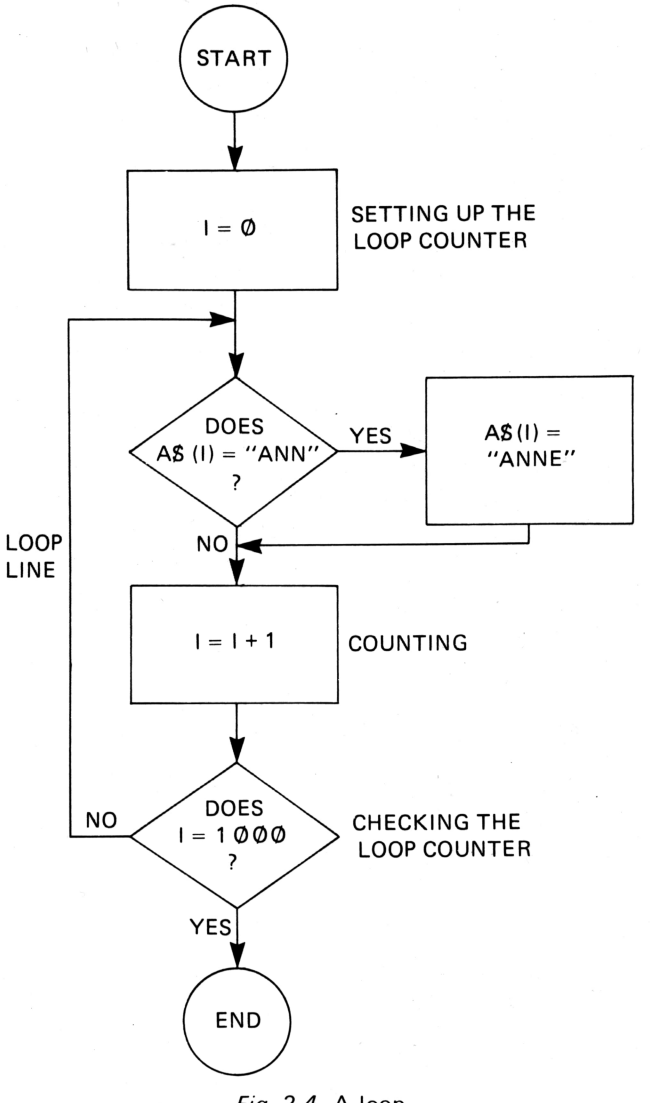


Fig. 2.4. A loop.

The chart would be extremely tedious to turn into a program, and would certainly not save any time. We need a way of mechanising this repetition of actions. This is performed by a *loop*, and is illustrated in Fig. 2.4.

This routine only has five program statement boxes, but you will see that it replaces the two thousand statements of the last chart. There are two concepts to grasp in Fig. 2.4: first, the concept of the *loop counter*, and second, the loop itself. The main loop comprises the two decision boxes, plus the statement:

I = I + 1

It sometimes also includes the statement:

A$(I) = "ANNE"

but only if the answer to the first decision is YES. Imagine the situation where the answer to both decisions shown in Fig. 2.4 were NO all the time. If you follow the arrows around the chart you will quickly see that you will go round the circle again and again. Each time this circle or *loop* is executed, the value of I, the loop counter, increases by 1. Thus, the loop counter I simply counts the number of times the loop is executed.

The program proceeds from the start as follows. The loop counter I is started off at 0, in the first assignment statement. We say that this sets up the loop counter. In the first decision box, the question is asked: 'does A$(I) = "ANN" ?' Since I equals 0 at this point, the implication is that we are really asking: 'is A$(0) equal to "ANN" ?' because I is meant to be replaced by its value wherever it occurs. If the answer to this question is YES, then A$(0) is misspelled, and must be changed to ANNE. The YES branch from the decision box is thus taken, and the assignment performed to correct the error.

The next statement which is encountered increments the loop counter by 1. This is shown in a common type of computer statement. The LET is deliberately left off here to illustrate the importance of understanding the real meaning of the = sign in computing. The statement:

I = I + 1

is a command, not a statement of fact - it is obvious that I and I+1 are never equal. This statement is used to change the value of I to I+1. Thus, on this first pass of the loop, I is changed from 0 to 1.

The next box is a decision, and asks whether I has reached the value of 1000 yet. As this is not the case, the NO branch is taken, and the loop line brings execution back to the first decision box. This time, A$(1) is checked, and again the correction made if necessary. The next box increments I to 2, and the whole process is repeated for this new value. Thus by incrementing I, all the variables in the list are checked, and any which are misspelled corrected.

Eventually, A$(998) and then A$(999) are checked. These are the last two in the list, because the list started at A$(0), and 0 to 999 inclusive is 1000 numbers. After A$(999) has been checked, I is incremented to 1000. The second decision box is then executed, and as I = 1000 the answer is YES, and the program finally ends.

The whole array of names has been dealt with in a very short time by a fast computer program, and the computer program itself would only take a minute or two to write by anyone familiar with BASIC. This saves perhaps hours of work by the method of Fig. 2.3.

This describes a simple routine, which may be part of a larger program. As you can see, every move of the action must be specified and written in

precise and clear terms using words which the computer can understand.

The final task of this chapter is to return once again to the problem of rearranging numbers into ascending order. The next section shows you how to perform this task using a flowchart.

Numbers in ascending order

The problems of rearranging three numbers in ascending order is a special case of an important problem in the processing of large quantities of data. For instance, a glance at any phone book will underline the necessity for arranging the data there in order, in that case in alphabetical order. Even though a phone book may have many hundreds of thousands of subscribers, it only takes a minute or so to find any one. This task would be almost impossible if the entries were not set out in a familiar order.

It is a common function to require data to be sorted into some sort of order, and the facilities of database programs such as Archive allow this to be performed with ease. Sorting can be performed in a number of ways. The

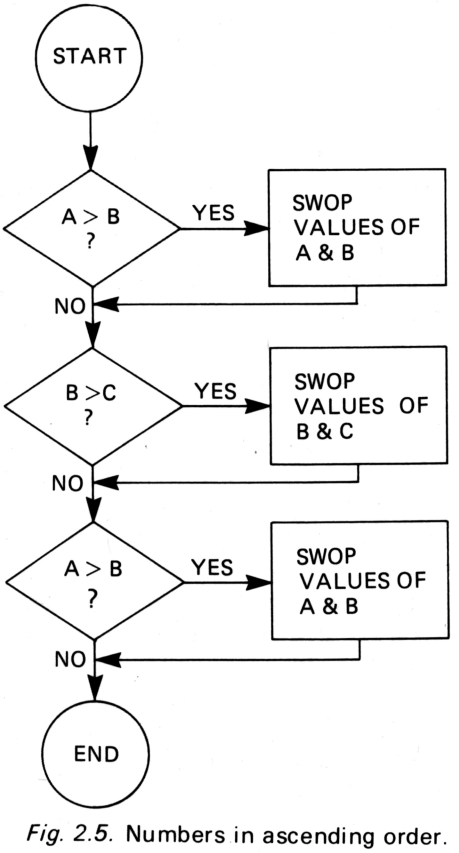


Fig. 2.5. Numbers in ascending order.

method used by the flowchart of Fig. 2.5 to sort three numbers stored in variables A, B and C into ascending order is a special case of a *bubble sort*. Note that a new sign has been used in this flowchart, it is the *greater than* sign >. Thus A>B is pronounced as 'A greater than B'. It can be a statement of fact, or a question, and depends upon the context. Here it is a question.

A, B and C are identifiers, and their values will change through the routine until the values of A, B and C are in ascending order with A less than B less than C. We can take an example of three values to illustrate the activity as the process proceeds. Suppose the numbers are:

A	B	C
9	3	1

These numbers are in completely the wrong order. The first decision in Fig. 2.5 asks whether A is greater than B. If it is, it must be swopped, as we want the smallest number in A, and not the largest. Thus the numbers become:

A	B	C
3	9	1

The next decision is to compare B with C, and swop if B is greater than C. This gives:

A	B	C
3	1	9

We now have the largest number in C, which is the start of an ascending order. Notice how the 9 has *risen* to this position from starting in A. We say that it has *bubbled* to the top, and hence the name *bubble sort*. The process is sometimes also called a *stone sort*, as the smallest number drops to the bottom like a stone, and will end up in A. The next operation is, in fact, to compare A and B again, and swop if A is greater than B, which it is here, and the program ends with:

A	B	C
1	3	9

as required.

You should try this routine out with your own set of numbers, perhaps including two numbers which are equal, and then using a set which is already sorted correctly.

Notice that Fig. 2.5 is an example of a repetitive process, and imagine what the program would be like if there were 1000 numbers to be sorted into ascending order. The process looks as if it should be able to be performed using a loop, rather than writing out decision after decision separately. In fact, it is possible to use a loop, and Chapter 6 will give this case as an example, both for numbers and strings in general.

Summary

- To manipulate quantities, or *variables*, by computer, they must be given names, or identifiers. An identifier must start with a letter, but can contain letters, numbers and underscore symbols. Up to 255 characters may be used in an identifier. It is also helpful if identifiers are meaningful but also as short as possible.
- To give a variable a value, an assignment statement must be used. This is of the form:

  LET A = 123

  or

  A = 123

- All keywords must have a space after them unless followed by an item whose first character is not a letter, number or underscore. Otherwise, the keyword will be considered as being part of an identifier containing the keyword, along with anything which appears after it, up to the next space or special character.
- Strings of characters must be enclosed by quotes, or *literals*, and their identifiers must end in a $ sign. Thus:

  A$ = "This is a string"

- To display the value of a variable on the screen, the PRINT statement is used. For example:

  PRINT A$

  This would cause:

  This is a string

  to be printed after the last assignment statement above.
- To arrange a group of variables numerically, subscripts may be used on the identifiers, but they must be enclosed in brackets. Variables of this sort are called *array variables*. An array of strings would look like:

  NAME$(1), NAME$(2), ... , NAME$(24), NAME$(25)

  Each of these variables contains a string, and the numbers in the brackets can also be replaced by a variable, such as a loop counter.
- Loops are processes which act repetitively on a set of variables. The number of times for which the loop has been executed is monitored by the value of a loop counter.
- Flowcharts give a means of setting out the processes within a routine in a visual form which is easy to transmit to other people. If you do not actually use a formal flowchart when you are programming, you should at least set down the main ideas on paper in this way. You will find that it

helps to structure your programming in a logical and efficient manner. It also leads to a more organised pattern to your programs, and this is of considerable help when seeking *bugs*, or programming mistakes.

Chapter Three
Starting with SuperBASIC

In this chapter we will see how some of the most important BASIC words are used, and we shall look at the ability of the computer to act as a super calculator.

One of the central concepts in computing is the idea of the *stored program*. In the last chapter, a couple of program lines were typed in, and we saw how they were executed immediately. Here we will see how program lines can be stored for later execution. This allows you to build up large programs which use the SuperBASIC language to its fullest extent.

As the BASIC words are introduced, examples will be given to allow you to grasp the ideas of writing and editing programs, and using the keyboard and screen of the QL to their utmost. This chapter serves as an introduction to the main facilities of the QL, and will be referred to throughout the rest of the book.

Stored program lines

The last chapter introduced the program as a sequence of commands to the computer, and flowcharts were used to explain some of the main ideas of programming. However, to construct a program, we must have a method of writing and storing the entire sequence of statements before actually executing the program. It is no good simply executing each statement as it is written before the program is completed.

One of the most crucial points is that program statements must be executed *sequentially*. This is very much like ordering a set of numbers. We wish the statements to be executed from first to last in a definite order. Many computer languages, including SuperBASIC, achieve this sequentiality by labelling each program line by a number. The computer then simply executes the program statements from first to last in ascending numerical order.

We can give each program line a number simply by typing in a number at the beginning of the line. When the computer sees a number at the start of a line, it simply stores the line in memory, using the number as a label. It does

not execute the statements on that line until it is told to do so by another command later on. For instance, the two-line program in the last chapter could be written as:

```
10 A$ = "This is the QL!"
20 PRINT A$
```

The computer automatically stores these lines, and waits for you to tell it to execute them. Any numbers can be used for labelling the program lines, as long as they are in the order of execution of the lines. Try typing these lines in, remembering the ENTER key at the end of each line, and the space after the keyword PRINT. All the other spaces are superfluous, but you are recommended to use them throughout for neatness. If you make certain types of mistake in typing in this program, you will find that the computer is still checking that the grammar or *syntax* of its language is being used correctly.

The above is an example of a stored program – perhaps your first ever! This program is stored in memory, and displayed on the screen with a white background to show you that it is not in the output window. If you have a video monitor, it will appear in the white window anyway. Notice how, as you type in, the program lines are input at the bottom of the screen. They stay there until ENTER is pressed, and the computer is allowed to store them for later use. Only then do they become displayed at the top of the screen.

Editing

If you make a typing error in a line, and you notice it before you press ENTER, you can alter it using the rules described in Chapter 1. If you do not notice the error until the line is displayed at the top of the screen, you can alter it by retyping that line. For instance, if you now type in:

```
10 A$ = "Hello!"
```

and press ENTER, the old line number 10 will be replaced by this new line number 10. The old version will have disappeared completely.

Another important editing feature is the one that allows you to add more lines to the program, anywhere in the sequence. For instance, suppose you want to add a program line to be executed between lines 10 and 20 above, you simply choose any number between 10 and 20, and use this to label the line. For instance, type in the following:

```
15 B$ = "This is the extra line."
```

When you press ENTER, you will see the line inserted into the correct place on the screen. This shows you a good reason for always numbering program lines in 10s. It leaves plenty of space between for inserting more statements.

If you wish to remove a line, you simply retype that line number, but leave it empty. Try typing:

```
15
```

When you press ENTER, the computer recognises that you have typed a line number 15, which is already stored, and will replace the present line with whatever you write. It then notices that there is no data in the line, and hence replaces the old line 15 with an empty line. However, the computer never displays lines which are empty, and so it will remove line 15 completely.

To scrub out the current program, and start with a clean slate, simply use the BASIC statement NEW. This will remove all the current program lines, and leave you to start again. (Do not do this yet!) An associated BASIC word is CLEAR, which clears all the variables which are in use at any time, and will be used later.

These operations give a quick simple method of program editing. We will see later on in this chapter that there is another, more sophisticated method of editing on the QL.

Running the program

Execution of the program is called *running*. We say that a program is run, and this is exactly how the command is phrased in BASIC. Try typing the following:

```
RUN
```

As there is no line number, this BASIC statement is executed immediately upon your pressing ENTER. This causes the program to execute. If you have followed the operations above, the screen should show:

```
Hello!
```

Because the program has been stored by line number, and not simply typed in and executed immediately, we still have the program in memory to change and re-run at will. If you are using a TV, you will find that the output of this program may have overwritten part of the program on the screen. However, the memory image of the program has been left untouched. Try typing RUN again a few times. As the program runs over and over again, the screen fills with the word: Hello! Keep going until your TV screen shows none of the program at all, and is filled with the red output window. If you are using a video monitor, you can display both program and output on the screen, and only the right-hand window will be affected.

Listing the program

To display the list of program lines back on the screen, you use another

BASIC word: LIST. Try typing this and see the effect. If you have a TV, the program will reappear, though it may take some sorting out if you have filled the screen with program output. If you are using a monitor, the program will be repeated again beneath the last display. You can always use LIST to see the stored program. If the program is too long to fit on the screen, you can list just part of it if you wish, as we shall see.

Arithmetic

One of the most important tasks that a computer can perform is arithmetic. Later on we will see how remarkably powerful the QL is at calculation. However, let us first see how the computer performs simple calculations, using just the operators of addition, subtraction, multiplication and division.

To perform addition and subtraction, you type in the operations just as you would into a calculator. The only difference is that you have to use a PRINT command to tell the computer what you wish it to do with the answer. If you do this, the answer will be displayed on the screen just as on any normal calculator display. For instance, try typing in:

```
PRINT 123 + 456
```

As this has no line number, it is executed immediately, and the answer, 579, will be displayed in the output window at the bottom of the last output - it may be difficult to see. You can always use the computer in this way, as BASIC recognises ordinary numbers, and the + and − signs. Try typing:

```
PRINT 579 − 123
```

The answer 456 should appear after you press ENTER. Try a few more calculations, perhaps using some negative numbers, and numbers with decimal points. All these calculations will result in an immediate answer on the screen, and once again the program listing will disappear from a TV screen.

Multiplication is performed as above, but the symbol used is not ×, because this becomes confused with the letter of the alphabet. The universal BASIC symbol for multiplication is asterisk *. Now try typing:

```
PRINT 12 * 10
```

The answer 120 will appear. Try a few more calculations using * and watch the results. Again, this is exactly as for a calculator.

To divide two numbers, the oblique symbol / is used. Thus type:

```
PRINT 12 / 10
```

and see the answer 1.2 appear. Now try typing in calculations using / to practise the idea.

Order of evaluation

When several of the above operations are combined, some extra rules have to be learned. For instance, if you want to evaluate the following *expression* or series of calculations:

```
12 + 10 * 8
```

you will have to know in which order the various parts of the calculation are to be performed. If you type this into most simple calculators, you will find that the answer given will be: 176. Try typing the above into the computer preceeded by a PRINT, and see what it gives. The answer is 92 on the QL. This is not due to any inaccuracy in either machine, it is because the two machines are performing the calculations in different orders. The calculator will add 12 and 10 first, and then multiply by 8, while the computer will multiply 10 and 8 first, and then add 12.

There are some rules which you have to learn to tell you the order in which BASIC evaluates a series of calculations. The rule is that all multiplications and divisions are performed first, from left to right, and then additions and subtractions are performed from left to right.

For instance, in the expression:

```
10 + 10 / 5 * 6
```

the multiplication and division are performed first, from left to right, i.e. the / comes first, then the *. The addition is of lower priority, and is performed last by the computer, to give 22. Again, this will disagree with most calculators. Furthermore, if the / and * operations were performed from right to left, then 5 times 6 equals 30, and 10 divided by 30 is 0.3333, which when added to 10 does not give 22.

It is very restricting to have to juggle an expression around to ensure that it is executed exactly right, and you may specifically want the * to be executed first in the above. You can change the normal order of execution by using brackets. *The overriding rule is that the interior of brackets is executed before anything else*. Thus if the above is written as:

```
10 + 10 / (5 * 6)
```

then 5*6 is executed before anything else. The rest follows the rules stated, so 10 is then divided by 30, and 10 added to this to give 10.33333.

As you can see, the order with which an expression's calculation is performed is most important. The rules above are purely a convention which we will stick to from now on.

Some examples

Try typing the following, and see if you can work out the answer before you press the ENTER key each time:

```
PRINT 15 - 3 * 3
PRINT 15 - 3 / 3
PRINT 3 / 3 - 3
PRINT 18 - 5 + 6 * 2 + 1
PRINT 100 * 0.5 / 25 + 20
PRINT 10 * (0.5 / 25) + 20
```

Calculators give different answers for some of the above because they usually execute each calculation immediately upon entry. They cannot normally store the operations for later execution. How many of the above 6 calculations would be the same on a simple calculator?

To use these arithmetic operations in programs, we will often have to use variables to store the numbers in the expression, as well as the answers. For instance, try typing in NEW, to clear the memory of any previous programs lines, and then type in the following program:

```
10 A = 10
20 B = 20
30 C = 2
40 D = A + B * C
50 PRINT D
```

Remember that the only space you have to remember is the one following PRINT, and that the variable identifiers can be in upper- or lower-case letters, and the computer sees them as the same variables. PRINT can also be in upper- or lower-case, or a mix of the two. If you try this, you will find that the computer changes it to upper-case in the program list.

When you run the program by typing RUN, followed by the ENTER key, the answer 50 will appear. Remember that this program can be run again and again. Also, you can type LIST at any time, and see the program listed out on the screen.

After running the program, the values of the variables remain in the machine. Thus, if you now type in:

```
PRINT D
```

The value of D, which is 50, will be displayed again. If you repeat this statement, 50 will appear as many times as the statement is typed, showing that the value of D is not affected by being viewed in this way. You can view the values of any of the other variables in the same way. This is very useful in trying to find out what has happened in a complex program, which may contain a bug. You simply stop the program, if it does not stop of its own accord, and print any of the variables which you think may give you a clue. You will see how to stop a program and perform this action later. The variable values can be cleared by typing in CLEAR. Try this now, and then

try printing D or any of the other variables. You will find that the computer does not simply print 0, it actually tells you that these variables have not yet been set equal to any number, by displaying an asterisk.

Strings

You will remember that a string of any characters can be stored in the computer, using identifiers with a $ sign as the last character. It is also possible to use the operation & between strings. This simply *concatenates* or combines the strings into one large string. For instance, type in NEW, followed by:

```
10 A$ = "HEL"
20 B$ = "LO"
30 C$ = A$ & B$
40 PRINT C$
```

When you run this program, the first two lines assign strings to A$ and B$, and the third stores their combination in C$, by combining them from left to right. The PRINT statement then gives HELLO on the screen. If you now type in:

```
30 C$ = B$ & A$
```

and run the program you will see that LOHEL is displayed, showing that the order in which the strings are combined is important. This is not the case for addition of numbers, for instance, but does apply to subtraction.

A number can be treated either as an arithmetic quantity which can be acted upon by the four arithmetic operations of the last section, or as a string of characters. For instance, type in:

```
10 A = 12
20 B = 15
30 A$ = "12"
40 B$ = "15"
```

and type RUN. Nothing will appear on the screen, but the assignments above will have been performed. Lines 10 and 20 put 12 and 15 into A and B respectively. Lines 30 and 40 put the characters 1 and 2 into A$, and 1 and 5 into B$. The literals, or quotes, imply that the characters are simply copied into the variables without any intelligence, and should not normally be recognised as numbers.

Now try typing in:

```
PRINT A + B
PRINT B + A
PRINT A$ & B$
PRINT B$ & A$
```

The four immediately executed statements above will give three answers. In the first two statements A and B are treated as numbers, and the answer is 27. This is because it does not matter which way round you add two numbers. The third statement will give:

```
1215
```

and the fourth statement will give:

```
1512
```

The numbers in the string variables are not regarded arithmetically when combined with the string operator &; they are just bunches of characters to be copied verbatim.

In the majority of computers, strings containing pure numbers as above will not be recognised by the arithmetic operations, even though they appear to contain numerical information. Arithmetic operations between strings will normally cause an error to be put up on the screen. However, the QL has a special facility called *coercion* which intelligently compensates for this, and automatically recognises what you are trying to do. For instance, try typing in:

```
PRINT A$ + B$
```

Even though these are strings, the QL recognises the + sign as a numerical operator, and tries to convert the strings to numbers, before applying the addition. The answer will thus be 27.

Data types

There are two main *data types* – numbers and strings. Strings are completely straightforward, and there is little more to learn about the actual data which can be stored. Numbers on the other hand are a little more involved. We have to know the range and accuracy of numbers which can be stored. It is no good trying to multiply numbers having 15 places of decimal if the machine can only calculate to 5 or 10.

Numbers in the QL are stored in one of two forms. All the numbers so far have been naturally assumed to be in *floating point* format. This means that if a decimal point were needed it would simply be put into the correct place, and float around as the calculation proceeded. The computer automatically puts the decimal point in the correct place. Most calculators do the same.

The other form of storage is called *integer* storage. Integers are never allowed to have a decimal point, and hence are confined to being whole numbers. They may be positive or negative, and they range from −32768 to +32767 on the QL. As you probably know, when a number is positive, its sign is usually omitted, but if negative, a minus sign is placed before it. This is called the *unary minus*. The QL uses this convention throughout for both

floating point and integer numbers. Integer variables are distinguished within the computer by tagging a % sign on the end of their identifiers. For instance, NAME% or D% or This_is_an_integer% will all do as integer variables. The % stops the computer from allowing the decimal point to be used for these numbers, and they are rounded off to whole numbers by the computer in calculations.

The range and representation of floating point numbers is more involved, and we shall look into this now.

Floating point numbers

The QL normally represents numbers in floating point, unless a % is used, and for this section we shall assume that all the numbers are in floating point format. This section is a little technical, and if you are not interested in manipulating very large or very small numbers, you can skip the section, and come back to it later when you need it.

The QL performs its calculations to 8 significant figures, but only displays its calculations to 7 significant figures, with the last number rounded off. This means that, for instance, if 10 is divided by 3, the QL will display 3.333333 as its answer. This is the answer to 7 significant figures of the division. This particular division should actually give an answer which has 3s going on to infinity, and any type of calculator has to decide how far it is going to go in representing such numbers. Seven figures is quite accurate, and there are few calculations which require more. If a number is not needed to the full 7 figures, then it is truncated, or shortened accordingly. For instance, the answer to 10 divided by 2 is simply 5. It would be wasteful to print this each time as 5.000000, and the computer will simply give 5.

It is important to be able to express and manipulate very large numbers too. To make them easier for humans to handle, a special convention has grown up, and is generally used in BASIC. It is called the *exponent* form. To see how this works, consider how you would write the answer to the following calculation:

```
10,000 * 10,000 * 10,000
```

The answer could be written as:

```
1,000,000,000,000
```

which is 1 followed by twelve zeros, which we call one thousand billion (American style). This is very cumbersome, and we are prone to make mistakes in reading and writing this type of number. The problem becomes even more acute if the number is:

```
9,436,894,607,987
```

Even saying this is not easy. It is nine thousand, four hundred and thirty six

billion, eight hundred and ninety four million, six hundred and seven thousand, nine hundred and eighty seven!

Both of the above numbers can be better expressed in a condensed form, using powers of 10. This requires you to understand or learn the following table:

10 to the power 0 (10^0) = 1
10 to the power 1 (10^1) = 10
10 to the power 2 (10^2) = 100
10 to the power 3 (10^3) = 1,000

and so on. *Raise to a power* means multiply the number by itself a number of times equal to the power. Thus 10 raised to the power 12 means multiply 10 by itself 12 times. Note that the number of zeros after the 1 above is equal to the power to which 10 is raised. Thus one of the large numbers above can be expressed as:

1,000,000,000,000 = 10 to the power 12.

There is a shortened form of this in BASIC, and it is:

1E12

This means 1 times 10 raised to the power 12. The number 2,000,000,000,000 is simply 2E12, which means 2 times 10 to the 12th.

More complex numbers can be expressed in this form too. For instance, 2,269 could be expressed as 2.269E3, because if you multiply 10 to the 3 by 2.269, the answer is:

2.269 * 1,000 = 2,269

The very large and complex number above (9,436,894,607,987) can be expressed as:

9.436894607987E12

If nothing else this is easier to say! It is normally said as nine point, then a list of digits ending up with E twelve. However, the QL cannot store numbers to this accuracy, and the number is truncated or rounded off to 9.436895E12.

The notation is also useful for the storage of very small numbers. For instance, 0.001 could be written as:

$$\frac{1}{1000} = \frac{1}{1E3} = 1E{-}3$$

Note that 1 divided by 1E3 is the same as 1E−3. This is a mathematical law, and simply has to be remembered. Another example of this type of number is:

$$0.00000023456 = \frac{2.3456}{1E7} = 2.3456E{-}7$$

The rule is that the number on the extreme left of this equation is changed to 2.3456 (a number between 1 and 10) by moving the decimal point 7 places to the right. This is the origin of the 1E7 above.

The QL, using the exponential notation above, can store floating point numbers in the range:

1E−615 to 1E615

This range is so great as to be quite meaningless. If the two numbers were to be written out in full, they would take up about a half a page of this book.

As an example of calculating with large and small floating point numbers, try typing in:

```
PRINT .00075683 + 8.5E6
```

The answer to this will illustrate how illogical it is, for instance, to add a very small number to a very large number. Try typing in:

```
PRINT .00075683 * 8.5E6
```

This gives a reasonable result, and in general you can multiply large and small numbers.

We shall see some examples of floating point calculations later, but before we can use the computer to its fullest extent, there are a few more important facilities which should be learned.

AUTO and RENUMber

These two commands are designed to help in program writing. When you are typing in a long program, it is useful to allow the computer to generate the line numbers automatically at the start of each line.

If you start your programming session by typing in AUTO followed by ENTER, then every time you type a line, and finish with an ENTER, the computer will start a new line for you by printing the next line number. The first number will be 100, and successive numbers after that will be at intervals of ten, as for the programs we have seen so far. You can also alter the starting number and interval, by typing numbers after the AUTO command. 100 and 10 are called the *default* values of AUTO, because if you default in typing numbers after AUTO, these numbers are assumed. The first number typed after AUTO is the start number, and the second the interval. Thus:

```
AUTO 120, 20
```

will start at line number 120, and number successive lines at intervals of 20. If you leave out the second of these numbers, the interval will automatically default to 10. Just as for all other keywords, AUTO must be followed by a space in this situation.

If you suddenly discover that one of the previously typed lines needs correcting, or if you have finished the program, you can exit the AUTO mode by typing CTRL SPACE, which means press the CTRL key, and while holding it down, press the SPACE bar. This pair of key pressings is called BREAK, and will also be used to stop programs while they are running.

Try entering AUTO mode, and type PRINT followed by the ENTER key several times. This will illustrate the point adequately. Now try some other starting number, and some other interval. A little practice will make the use of AUTO quite clear. Exit the AUTO command, but do not erase the program lines you have typed in, they will be useful for the next part of this section.

As you correct a program, you will find that you have to insert extra lines, and these will be out of sequence with the normal line number intervals. This does not matter, but you may even find that you have to insert so many extra statements between some lines that the interval is not big enough to contain them. There are many reasons why you may wish to renumber the whole program, or part of it, during its development. The RENUM function is used for this purpose.

If you simply type in RENUM, followed by ENTER, the whole program will automatically be renumbered from 100 upwards at intervals of 10. These are the default values of the RENUM command. Try inserting some extra lines between lines of the program left over from the first part of this section, and type RENUM. This will have no immediate effect on the program lines displayed, but will have affected the memory image of the program. To view the newly renumbered version of the program, type in LIST. This will list the program starting from the bottom of the last display, and scrolling the old lines up the screen if necessary. You may have a little trouble sorting out the new material. Type LIST again, and watch the screen carefully as you press ENTER.

If you wish to renumber just a part of the program, and wish this part to start at some specific number, and increase at intervals other than 10, then all these parameters can be used in the RENUM statement. The most general form of the statement is:

```
RENUM A TO B;C,D
```

A is the number of the first line to be renumbered, B is the last. C is the first line number of the renumbered version, and D is the increment. The 'TO B' can be omitted, and the default value will be the last line number of the present version of the program. In other words, it will do all the line numbers from A to the end of the program. If D is omitted, the default value is 10, and this will be used as the increment. The new version will start at line number C. Also, note that RENUM and TO are BASIC keywords, and each requires a space after it, as they are both followed by letters.

The only way to see how RENUM works properly is to try it. Try using all

the parameters in the RENUM statement on the present program stored in memory, and try to predict what it will do before you press ENTER – watch the screen carefully as you press ENTER. Keep trying until you can predict correctly. This is a very useful command, and you will find yourself using it more and more as you write longer and longer programs.

LIST, delete

LIST can be used to LIST part of a program, by adding some numbers after it to tell the computer which lines to display. For instance, if you type in

```
LIST 10
```

just line 10 will be displayed. Similarly,

```
LIST 10,50,60
```

will just list lines 10, 50 and 60. Try this out by renumbering the program you have stored from the last section, to start at 10, and increment by 10, or input a whole new set of lines to practise on.

If you have a number of program lines stored, you will be able to LIST a section of the program by typing

```
LIST 10 TO 50
```

This will LIST all lines from line number 10 to line number 50 inclusive. If you do not have any lines in this range in your program, try changing the parameters above, to practise the use of LIST.

You can leave out one or both of these numbers, and a default value will be assumed by the computer. For instance,

```
LIST TO 50
```

will LIST all the lines from the start of the program to line 50, inclusive, and

```
LIST 50 TO
```

will LIST all the lines from 50 to the end of the program.

You can mix the two types of parameter in LIST by typing

```
LIST 10,30,60 TO 100
```

This will display lines 10, 30 and 60 to 100 inclusive. Try this out for yourself, and become familiar with LIST – it will be used a lot.

This is not quite the most general form of the LIST function. The complete version will be explained in a later chapter, where you will see how to use LIST to output program lines to a printer or any other peripheral.

As well as listing a group of lines for inspection, you can delete a group of lines using the DLINE command, followed by numbers just as for LIST. For instance.

```
DLINE 30,50, 100 TO 200
```

will delete lines 30, 50 and 100 to 200. This saves having to type each line number followed by ENTER to delete a large number of lines at any time.

RUN

Another useful facility concerns the RUN statement. It can be used on part of a program. You do not have to RUN the entire program from the start. If you type a number after RUN, it will only RUN the program from that line number. For instance, type NEW on the keyboard, followed by pressing ENTER, and you will wipe the program you have been using. Now type in:

```
10 PRINT 10
20 PRINT 20
30 PRINT 30
```

When you RUN this, you will see that 10, 20 and 30 are printed on separate lines. Now type in:

```
RUN 20
```

This will cause the program to RUN from line 20 onwards, missing out line 10. Only the last two numbers will be printed. Now type in:

```
RUN 30
```

and only the 30 will be printed.

RUN can have a numerical expression after it too, so that, for instance,

```
RUN 5*6
```

will cause line 30 to be RUN.

REMarks

Chapter 2 introduced a method of working out and documenting a program using a flowchart. This enables the writer, or anyone else, to understand the logic of a given program at a later date. As mentioned there, another method of explaining the working of a program is by putting comments into the program itself. If every main routine in the program had a heading to tell the reader about it, this would help tremendously in its understanding. However, if you simply type some text into a line of the program, the computer will try to interpret this as a program statement, and display an error.

If a program statement begins with the BASIC word REM, then the computer ignores everything which follows on that line, but faithfully prints it out in a LIST. This allows you to comment the program in any way you

wish. You are recommended to do this liberally, as you will find that you will forget how your program works in a very short time. Well-placed comments will help you to re-read the program later. REM will be used in programs in this book to help you understand the examples which are given. The last program could have had a comment stored in it as follows:

```
5 REM This prog. can be used for RUN experiments
```

You should type this in as a continuous line of text. The computer will not be able to contain the whole statement on one of its lines either, but it will break it up at some other place. Note that REM is expanded to REMark in the computer's screen listing.

You could add another line after this REM to space it out a little. For instance:

```
7 REM
```

You may find that it helps readability in a long program.

Loops

The last chapter introduced loops in a pictorial form, and this section introduces the simplest BASIC loop statement which exists. It will help you to repeat programs without having to retype RUN over and over again. The BASIC word is GOTO followed by a line number. It simply tells the computer not to execute the next line in the program, but to follow this instruction, and go to the line which is indicated in the statement. To use the example given below, you will have to use the BREAK from the keyboard to stop the program, as it will simply go on forever or until you switch off. BREAK is accessed by holding down CTRL and pressing SPACE.

Use NEW to start the program from scratch. Then type in and RUN the following. You could use AUTO to save you typing each line number from scratch, and you can BREAK when you are finished.

```
100 REM *** SET UP LOOP COUNTER
110 N = 0
120 REM
130 REM *** LOOP STARTS HERE
140 REM *** PRINT NEXT VALUE
150 REM *** IN 12 TIMES TABLE
160 REM
170 PRINT 12*N
180 PRINT
190 REM
200 REM *** INCREMENT LOOP COUNTER
210 REM
220 N = N + 1
230 REM
```

```
240 REM *** NOW GO TO TOP OF LOOP
250 REM
260 GOTO 130
```

Notice that GOTO is printed by the computer as GO TO with a space as shown. You can still type GOTO without the space to save time. Remember that just as for all BASIC words, a space must follow GOTO as it is followed by a number.

This program prints out the 12 times table until the numbers become too big for the computer to handle. However, the process is comparatively slow, and it will take a very long time to overflow, as the top number possible is very high. There is no natural method of exit, and it relies on you to BREAK, and hence stop it forcibly. The empty PRINT statement in line 180 simply causes an empty line to be printed to separate the numbers a little more neatly than using every line for output.

As you can see, the program is liberally supplied with REM statements to give the reader some idea of what is happening. It is not suggested that you have to use quite so many empty REM statements, but they do help to make the comments stand out, as do the asterisks which have been included for the purpose. This small program probably does not actually need quite so much comment, but it illustrates the idea.

The GOTO statement can also be followed by a numeric expression. For instance, GOTO 2*65 would give the same effect as in line 260 above. Similarly, if the numeric variable LOOP had been set to 130, then GOTO LOOP would also work in this place.

GOTO can also be used as an immediate command. For instance, if you have pressed BREAK, and wish to continue, one way of doing so without resetting the variables is to type in:

```
GOTO 130
```

The program will continue with the values reached at the BREAK. In fact, CONTINUE is also a BASIC word, and will achieve the same thing without having to know exactly which line to use in GOTO. Also, when you BREAK a program, you may not be able to use GOTO to continue exactly where it broke off. CONTINUE does continue from the exact point at which BREAK occurred.

Looping is a very important function of a computer program, and we shall see in the next chapter how sophisticated SuperBASIC is in providing different methods of looping.

PAUSE

A method of slowing down the loop to see the individual processes occurring is the use of the PAUSE statement. This takes a number after it to tell the computer how long to pause when PAUSE is executed. The number

is in units of a fiftieth of a second (20 milliseconds). Thus a number of 50 would give 1 second, 300 would give 6 seconds, and so on. Try adding in the following line to the program above:

```
255 PAUSE 200
```

This will considerably slow the program's display.

A key pressing on the keyboard will stop the PAUSE, and continue the execution. Try this during one of the 4 second PAUSEs above.

A PAUSE statement without a number will halt the program forever unless a key is pressed. Try removing the 200 above, and see how the keyboard can be used to decide when the next execution of the loop will be performed.

Multiple statement lines

So far, each program line has contained just one statement. Nowhere have we placed two PRINT or assignment statements on the same line. There are advantages in being allowed to put more than one statement on a line, and it creates a more compact program when you do. To separate statements on the same line, you must use a colon (:) between each statement, so that the computer will know when one statement ends, and the next starts. For instance, the following is a valid BASIC statement, and you can use it in the immediate execution mode, or in an actual program. Note that colon (:) is not a BASIC word, and thus does not require a space.

```
A=10:B=3:C=4:PRINT A*B*C
```

There are four statements here, three assignments and a PRINT statement. This line will set A, B and C to the values shown, and then print their multiple.

As an example of compacting programs, the 17-line program of the last section could be written, without REMs, as:

```
10 N=0
20 PRINT 12*N:PRINT:N=N+1:GOTO 20
```

Note that there is no space after the second PRINT. The colon is a perfectly good method of separating the PRINT from the next character which might otherwise be confused as being part of an identifier including the letters of PRINT. Colon, you will remember, cannot be used in an identifier.

This program achieves the same as the earlier one. You do not even have to use NEW before typing it in. The program takes up line numbers which are numerically before those of the last program, and when you type RUN the computer will start executing at the lowest number line in the memory. This happens to be line 10 above. The computer then becomes stuck in the loop in line 20, and never manages to reach line 100 where the next program starts. To RUN the longer program, type in RUN or GOTO 100. If you type

in GOTO 170, or RUN 170, then N will not be set to zero in line 110, and any existing variable values will be kept and used in the program. Try all these combinations to see what happens.

Using colons in this way, can produce some very complex lines, and it is important to be able to edit these long lines without having to retype the entire line. This is covered in the next section.

EDITing

The QL provides a neat and powerful method of editing lines where it would be tedious to have to retype them. To enter Edit mode, type in EDIT followed by the line you wish to edit. This causes the required line to be brought from memory to the command line at the bottom of the screen. This can then be altered using left- and right-arrow keys in the usual manner. When you have finished, it does not matter where the cursor is: just press ENTER, and the editor replaces the old version of the line in memory with the new one. The edit mode is then exited.

Try using EDIT on the line 20 in the last section. You could change it to display the 13 times table instead, for instance. To do this, type in:

```
EDIT 20
```

The command line will then show the line, and you will have to move the cursor over to the 2 of 12 in the first PRINT statement, and change it to 3. Press ENTER and the displayed line in the screen list will change accordingly.

To edit a sequence of lines, you can specify an interval after the first number in EDIT. If you still have the last long program in memory, try typing:

```
EDIT 100,10
```

This will bring line 100 down for you to alter. If you have no alterations, simply press ENTER. This will not exit the edit mode as before, but will add 10 to the present line number, and bring that down instead. Each time ENTER is pressed, the next line in the sequence is brought down. If the next line in the sequence does not exist, it will be created, and you can type into it if you wish. The editor is exited by pressing BREAK as for AUTO.

This completes the first set of SuperBASIC words and functions which you will need for the rest of this book. The next chapter explains *input* and *output* from your programs, and the material introduced in this chapter will be used throughout the next to help set it in your mind.

Summary

- Program lines are stored by the machine if they start with a line number.
- A new line can be typed in to replace an old one by giving it the same line number. The computer replaces the old line with the new one. This can be used to correct mistakes.
- Programs can be run using RUN followed by an optional line number. LIST will display the whole or part of a program.
- BREAK may be used to stop program execution by pressing CTRL and SPACE simultaneously.
- DLINE is used to delete sections of a program.
- NEW can be used to remove the current program, and CLEAR wipes out the values of all variables.
- Arithmetic operators * (multiply) and / (divide) are evaluated before + (add) and − (subtract). Brackets are always evaluated first. Otherwise, evaluation proceeds from left to right. The & operator is used to combine two strings.
- Variables may store strings, floating point numbers or integers. String identifiers end in $, integers in %. All others are assumed to be floating point.
- AUTO can be used to supply the next line number automatically during program development. RENUMber is used to renumber whole or parts of a program.
- REM is used to place remarks in a program which are printed in a LIST, but disregarded otherwise.
- GOTO forces execution to the given line number.
- Several statements can be placed on a line separated by colons.
- EDIT followed by a line number is used to bring a line down to the command window where it can be edited.

Chapter Four
Input/Output

One of the most important functions of a computer is communicating with humans. The hardware and software required for this is called *the man-machine interface* or the *I/O*, which is short for *Input/Output*. Early microcomputers had to use switches and lights to communicate with the user. It was no easy job to write programs and read the products of the programs with such rudimentary interfaces. Today, a keyboard and a screen are standard I/O interfaces.

Any programming language has to contain a fairly sophisticated set of instructions which will allow all the ideas which we can program into the machine to be communicated to the user. This chapter is devoted to describing the SuperBASIC words and functions which allow you to use the screen and keyboard in as complex a manner as you wish. A later chapter shows you how to draw coloured shapes and patterns on the screen.

The QL also contains an electronic circuit which acts like a quartz clock, and gives year, month, day, hour, minute and second. This chapter describes how this can be used and set by some BASIC statements, to allow the input of time to a program.

Output

The method of output for all the programming examples given so far is via the PRINT statement. Our next task is to expand the use of PRINT further.

Every time we have used PRINT in the above, it has output to a new line on the screen. It is often useful to print things on the same line, perhaps separated by single spaces. It can also be useful to arrange the printing in columns on the screen, or with successive items next to each other with no spaces in between. These different cases can be produced by some special *separators* used with the items to be printed.

To experiment with the PRINT statement, start off by typing in NEW, followed by ENTER, unless you have just switched on. This will clear the computer ready for a new program. Now type in:

```
10 A = 123
20 B = 12.998
30 C = 5
40 D = -13.2
50 E$ = "The"
60 F$ = "Answer"
```

Type RUN, and the program will execute. No output to the screen will be produced, but values will be assigned to the variables as shown. We can now experiment with the PRINT statement in immediate mode without interfering with the values above.

Separators

It is perfectly allowable to have more than one item after the word PRINT. Such a list of items is called a PRINT list. It is a rule that the items in a PRINT list have to be separated by one of four symbols or separators. These are:

, ; ! and \

If the items are separated by a comma, they are printed in columns of 8 spaces each.Try typing the following:

```
PRINT A,B,C,D
```

The result will be the values of these variables printed in 4 columns of 8 spaces each. Try typing this line a few more times, and the output will form four columns of numbers in a nice neat form. This is very useful for financial type programs, or lists of any other type of information. To increase the space between columns, two commas can be used to separate items, each comma will move the next item 8 spaces from the start of the one before. Try typing

```
PRINT A,,B,C
```

a few times to see the effect. If you add another comma between B and C, the printing overflows to the next line if you are using a TV.

Another useful attribute of separators is that if you finish the PRINT list with one, its effect transfers to the next PRINT statement which is encountered in the program. For instance, if you type in:

```
PRINT A,B,
```

followed by:

```
PRINT C,D
```

then even though the items are printed by separate PRINT statements, the display still prints them on one line, separated into columns. The comma at the end of the first statement tells the computer to continue printing as

commanded by the separator. Try typing this, and then try the same thing but without the comma at the end of the first PRINT list.

Items separated by semicolons are printed next to each other with no spaces. Thus, typing the following:

```
PRINT A;B;C
```

will give 12312.9985, and typing:

```
PRINT E$;F$
```

will give:

```
TheAnswer
```

with no space between the strings. This is very useful for printing composite strings from parts of others, and gives you complete control over where the spaces go. If you put more than one semicolon between two items, it acts the same as using just one. Once again, if you end the PRINT list with a semicolon, its effect transfers to the next PRINT, and the next object printed will be printed close up to the last item. Try typing in:

```
PRINT E$;
PRINT F$
```

This will give the same display as if E$ and F$ were in the same PRINT list separated by a semicolon.

When you use PRINT, you have already seen that ordinary numbers can be included in the PRINT list. You can also include strings using quotes. For instance, try:

```
PRINT E$;"Correct";F$
```

This will give:

```
TheCorrectAnswer
```

If you want to put in spaces, they could be included in the literals as:

```
PRINT E$;" Correct ";F$
```

This will make the printout more readable.

Now try typing in:

```
PRINT E$;" ";F$;" is ";A+B
```

This PRINT statement has 5 items to be printed; they form a PRINT list. The first is E$, which contains 'The', the second is a space between literals, the third is F$, with 'Answer' and the fourth is the word 'is' surrounded by spaces. The fifth item is a calculation. The computer prints the items in the PRINT list, from left to right, and, as you have seen, it will evaluate any expressions according to the normal rules, and print them in their turn. The answer to the calculation is 135.998, and this is printed with a message

formed by the previous PRINT list items. This shows you how useful the more sophisticated aspects of PRINT can be.

The exclamation mark is similar to the semicolon, except that rather than butting items together, it inserts a single space for each ! which is used. In addition, it prevents items from being chopped in half if they are about to be printed at the end of the line, and will not quite fit. The ! prints the item on the next line, without a space at the start. The ! symbol acts on the next item in the PRINT list, and is thus placed before items it is to act on.

Try typing in:

```
PRINT E$!F$
```

As you can see, this is a convenient method of saving the necessity of adding in a space between E$ and F$, as we had to in the above.

In addition, it can be placed at the end of a PRINT list, like a semicolon, and will act on the next item which appears in a PRINT statement. However, two !s are required, to produce a space.

Try the following:

```
PRINT E$!!
PRINT F$
```

Try leaving out one of the !s and see the effect.

To see the other function of !, try the following:

```
PRINT F$,F$,F$,F$!"This will overflow"
```

The 4 F$'s in the PRINT list take the printout very near to the end of the line. If the words in literals were to be printed now after a semicolon, they would be split between this line and the next. The ! prevents this, and takes the whole word to the next line. Try replacing the ! with ; to see the difference.

The backslash symbol \, is used to force printing to start on a new line. Try typing in:

```
PRINT E$\\F$
```

Each \ symbol causes a *line-feed* to be printed, which is the same thing as saying that a new line is printed for each backslash. You can also use an empty PRINT statement to give extra lines, as we shall see shortly.

You can now add some extra lines to the program stored in memory. For instance, A and B could be added together to give an answer, and C and D could be added to give another answer. These results can now be printed out in a neat and stylish manner using the PRINT statement. Try adding in:

```
70 PRINT "Addition of A and B gives"
80 PRINT E$!F$!"as:"!A+B
90 PRINT
100 PRINT "The second addition gives"
110 PRINT E$!F$!"as:"!C+D
```

Remember that every symbol and space in the PRINT statement has a precise and defined meaning. Look at line 80, and imagine the print-head of a typewriter. The first thing it types is the contents of E$. After this, the ! tells the head to print a space. The next item tells it to print the contents of F$, followed by a space again. Next comes three characters in literals, and another space. The calculation is performed, and the print head prints the answer. At this point it notices that the list has finished, and there is no other symbol to be considered, so it prints a line-feed, which simply returns it to the start of the next line.

Try to work out exactly what the program will display when it is RUN, and then run it to check you were right.

Try to add a few more blank lines in the output to space it out more, and make it more readable. Use one or more backslashes in lines 70, 90 and 100, for instance.

The AT statement

The above forms of PRINT only print items in a sequential fashion, and from top left of the page, down to bottom right, as we write in English. It is often very useful to be able to print an item starting anywhere on the screen. For instance, when we learn about the clock functions, we will see a program for displaying the time in the top right-hand corner of the screen, with the seconds actually ticking off. This requires a PRINT statement to be executed continually, and outputting constantly to the same area of the screen. All the versions given so far could start in the correct place, but would scroll the data down the screen, and not stay in one place.

The AT statement can be used to position the start of a PRINT anywhere on the screen. It regards the screen as a grid made up from the rows and columns in which PRINT statements output their characters. The actual grid differs between TV and video monitor. The AT statement has two numeric variables, and is placed anywhere in the program before the PRINT statement. However, for neatness it is often placed on the same line, separated by a colon. The numeric variables in the AT statement are COLUMN and ROW respectively of the required start of the PRINT. Thus, for instance,

```
AT 20,10:PRINT E$
```

will print 'The' at the 20th position on line number 10 from top left.

Try this out to gain a feel for the extent of the screen, and try to print F$ so that it ends in the top right position on the screen.

Clearing the screen

One other useful screen handling statement is CLS. When this is executed it

clears the part of the screen used for output. This is useful if you do not want the output to scroll continuously when executing a loop. For instance, type in:

```
10 PRINT "Hello"
20 GOTO 10
```

This scrolls the output continuously. Try adding in the following lines:

```
15 PAUSE 100
17 CLS
```

The screen will not fill up as the display fully clears before each 'Hello' is printed. However, the display is not good, and a better method of stopping the scrolling is to use the AT statement. In a later chapter you will see how to clear different parts of the screen with the CLS statement. For the moment, we will simply use it to clear the screen to allow a new program to start from scratch. Type in CLS to clear the screen whenever the display is full and needs clearing. This is particularly useful before a LIST.

Input

So far, all the data in the programs has been fixed within assignment statements. We will now see a method of allowing a program to take in data through the keyboard. This allows the program to interact with you while it is running, and allows you to control the values of the variables used by the program for its operations.

The BASIC statement used is INPUT. The INPUT statement contains a list of variables, very much like a PRINT statement. However, the values of the variables are not output to the screen, they are input from the keyboard.

As a simple example, type in NEW to clear memory if there is a program stored there, and type in the following program lines:

```
10 A=0.0001
20 INPUT B
30 PRINT A*B
```

The object of the program is to print the value of A times B. A is fixed by the assignment statement in line 10, but B is to be input by you when line 20 is executed. Try RUNning the program, and see what happens. You will see that the machine halts as if it were hung up, but with a cursor flashing in the output window. It is waiting for you to type a value for B. You can type in any number you wish at this point, and nothing is actually input into B until you press ENTER. This is the normal philosophy of the machine and should not be a surprise. If you make a typing error, or decide that you want to change the number, you use the normal methods of correcting the input. before pressing ENTER.

When you have typed in the value you wish B to take, and pressed

ENTER, line 30 is executed, and the value of A*B displayed.

You should RUN this program a number of times, using different values of input, to see the effect of using small and large numbers. If you input a very small number, such as 0.0000023, you will see how the QL is able to move automatically to displaying the answer in exponent form. You can also input numbers such as 4.5E10, and watch the result. The only rule to remember is that the data type of B is floating point numeric, and to ensure that the program executes correctly, you should only type this sort of input. Try typing in a string of letters and see what happens.

If you wish to input and display a string, you will have to use a string variable in the INPUT statement. For instance, alter the program by inserting:

```
25 INPUT A$
26 PRINT:PRINT
27 PRINT A$
```

Line number 25 adds a string INPUT statement. Line 27 adds another PRINT statement which displays the string typed in when executed. Try running the program. The first input required is the same as before, but the second input required is a string which will be printed verbatim. When the string is required, input the following words:

The answer to A*B equals:

This will add a message to the output. However, you have to input these words every time the program is run. It would be better to include another PRINT statement in the program to print out the words without the necessity of their being input by you each time.

Another useful addition to the general neatness of the output would come from some instructions to the user as to what to input and when. This can be done by PRINT statements before each INPUT statement explaining that you now have to input a number or a string. However, there is an easier way to do it, using the INPUT statement itself. You can actually place a message in the INPUT itself if you wish, and each time it is executed the message will appear just before your input is needed. The message is simply placed between literals, and written after the BASIC word INPUT. The message has to end with a PRINT separator such as a comma or ! after the last quote. Try typing in the following line:

```
20 INPUT "Please type in the value of B",B
```

When you RUN the program, you will see the effect, and how different it makes the general use of the INPUT statement. You do not have to use such long messages. For instance try:

```
20 INPUT "B=",B
25 INPUT "A$=",A$
```

and see how much easier it is to use the program when you run it. Try replacing the commas above with ! and ; to see the difference.

As an example of the use of the ideas introduced here, let us see how we would use a program to convert foreign currencies to British pounds. The program will assume that you know the rate, and that you want to convert a given number of currency units into pounds.

To solve this problem, we have to input the name of the currency, for later printing out, input the rate of conversion and then input the amount. The conversion is performed by dividing the amount by the rate. Finally, the answer is printed out. This short explanation could have been written as a flowchart, but is a little too simple for the effort. Before typing in the next example, use NEW and then CLS to give yourself a clear slate. A possible program for the conversion would be:

```
10 CLS
20 INPUT "Currency name is?"!CNAME$
30 INPUT "Rate per Pound is?"!RATE
40 INPUT "How much to  convert?"!AMNT
50 PRINT\\\\
60 PRINT "Currency conversion gives:"
70 PRINT
80 PRINT AMNT/RATE!"Pounds for"!AMNT!CNAME$
```

Before you run this program, spend a little time working out exactly what its output will look like. Most of the statements are straightforward, but you will have to analyse every single character in line number 80, and this will be a good exercise to practise some of the ideas presented so far for the PRINT statement. When you RUN the program, check that you were correct, and carefully analyse any errors you made. Remember that you will have to write statements such as this yourself when required.

The CLS in the first line of the program is a useful way of clearing the screen. The first three inputs in the program take in the currency name, rate and amount to be converted. The currency name stored in CNAME$ is a string, and it stores any characters you type in. You may choose dollars, or francs, but you may equally type in cats and dogs. The program would not know any different, and it would faithfully print it out in the answer as if it were a currency. This further underlines the fundamental law of computing. No computer program contains more intelligence than you have put into it yourself. This was stated at the start of Chapter 2 in the simple statement: *computers are stupid*. Be careful not to assume any more intelligence in the program than you have programmed in.

To take this example further, you could make it more intelligent on this one point of recognising a real currency name, simply by storing the name of every currency in the world in an array. When CNAME$ is input, the program could compare this value with each name in the array, and check that there is at least one which agrees with the name. If not, the computer could tell you that you have made an error, and ask you to try again. This

type of intelligence underlines another quite important fact about computers. To make them more intelligent, you often need a lot of memory!

The INPUT statement can also be used to input a number of variables. The variable names are placed in the INPUT statement, and separated by print separators. For instance, the program above could be written in a more compact form as:

```
20 PRINT "Input Currency name,"
30 PRINT "Rate per Pound, and amount."
40 INPUT CNAME$,RATE,AMNT
50 PRINT \\"You get"!AMNT/RATE!"Pounds
   for your"!CNAME$
```

It is very important that you type in the correct type of data for each of the inputs, and press ENTER *after each separate input item*. The inputs required are a string followed by two numbers. Anything else may upset the program, and cause an error message to appear. RUN the program, and experiment with different inputs to see where the program works, and where it goes wrong.

Other types of input

Another important form of input is achieved by writing the data into the program itself. We have already seen examples of this. Many of the programs we have seen so far simply assign numbers to variables as required. There is, however, a way of loading a large block of data into the program, and forcing the computer to work through the block, using it as it goes. This is useful, for instance, for storing a table of numbers which must be referred to during the program's execution. This type of input uses the BASIC words READ, DATA and RESTORE, as described below.

There are other types of input which can be gained by using the keyboard, and they are often used for programming the computer to play games. They allow an instantaneous key-pressing to be sensed by the computer. These use the BASIC words: INKEY$ and KEYROW. They will be introduced shortly.

READ, DATA, RESTORE

These BASIC statements allow a block of data to be written into the program, and accessed by the program during execution. The block of data is written, with commas as separators, using DATA statements. Any number of DATA statements can appear, and they can be placed anywhere in the program. For example:

```
10 DATA 1,2,3,4,5,6,7,8,9,10
20 DATA 11,12,13,14,15,16,17
```

To read these data into the required variables, the READ statement is used. This has a similar structure to INPUT. For instance:

```
30 READ A,B,C
```

The READ statement is followed by identifiers, separated by commas. When this READ statement is executed, the computer takes the values for the variables A, B and C from the first values in the first DATA statement in the program. The values are taken and used in the order in which they appear. Thus, A is set equal to 1, B to 2 and C to 3. Each time a READ statement is executed, the computer refers to the next values available, and continues through all the data in all the DATA statements until there is no more data left. Here, for instance, there are 17 numbers, and 3 variables to fill. This means that if the READ statement is in a loop, being executed over and over again, the READ would work 5 times, and then A and B would be set to 16 and 17 respectively, but C would not be able to be set, and the program will halt saying that it is at the end of the file of numbers. Try adding the following lines, and then RUN the program.

```
25 CLS
40 PRINT A,B,C
50 GOTO 30
```

As the reading occurs, the computer keeps count of the DATA statement in use at the time, and the exact position being accessed within that DATA statement. It uses a *data pointer*. You cannot read or control this data pointer yourself; all you can do is to use the RESTORE statement to restore the pointer's value to the start of whichever DATA statement you wish. If the program did not run correctly, try typing RESTORE. The data pointer may not have been at the top of the data blocks when run.

If you type GOTO 30 after running the last program, you will remember that the computer does not reset the states of the variables, and this includes the data pointer. The result will be the same error that stops the program naturally. In fact RUN does not restore the data pointer either – try it. However, if you type in:

```
RESTORE
```

The computer automatically resets the data pointer to the start of the first data statement, and the program can be run using RUN or GOTO 30. You should always start a program using READ and DATA with a RESTORE statement. Now type in:

```
27 RESTORE
```

This will allow you to RUN the program using RUN, from the start, more than once. You cannot place the RESTORE further down the program, or it

will restore the data pointer each time the program loop is executed, and only the first three numbers will be taken. The loop will also never end. Try it to see the effect.

If you want it to start at the second DATA statement, you can type in:

```
RESTORE 20
```

The RESTORE statement, therefore, can take a line number as shown, and the default value is the line number of the first DATA statement in the program. There is no formal limit to the number of DATA statements which you can write into a program, and no formal limit to the amount of data which can be put into a DATA statement. However, you should keep DATA statements manageable. If you split the data up into many DATA statements on different lines, then it is easier to access parts of the table by using RESTORE to one of the line numbers containing DATA.

Strings can also be stored and accessed in this way, by using the correct identifiers. For instance:

```
10 CLS
20 RESTORE
30 REM *** e.g. of numeric
   and string DATA
40 REM *** Read a string and a number
50 READ NAME$, A
60 REM *** Now print the message
70 PRINT "Name of number"!A!"is"!NAME$
80 REM *** Loop around until
90 REM *** data runs out.
100 GOTO 50
110 DATA "Mary", 1,"John",2,
    "George",3,"Anne",4
120 DATA "Keith",5,"James",6,"Sue",7
```

This program picks the data from the DATA statement in pairs again, but this time it takes in a string and a number. It is important that the data agrees with the types of identifier which are used in the READ statement.

RUN this program again, to see that the data pointer is automatically reset to the start. If you delete line 20, you will find that you cannot RUN the program more than once.

Instantaneous input from the keyboard

The INPUT statement waits for you to press a key before moving on to execute the next statement. In games and other programs where there is a lot of processing going on in the background, it would make the game less dynamic if it stopped dead each time input was to be requested from the player. To solve this problem, the QL offers two SuperBASIC statements

which allow you to input to the program dynamically, while not holding up the program when input is required.

The SuperBASIC word INKEY$ is used to allow a character to be input from any device connected to the QL. We will see in a later chapter how to make INKEY$ work with devices other than the keyboard. For the moment we will restrict ourselves to its use for inputting a character from the keyboard.

INKEY$ is an identifier for a special type of string variable. It has a value which is equal to the character on the key being pressed when INKEY$ is evaluated. To see how this works, use NEW to start a new program, and CLS to clear the screen. Type in the following:

```
10 PRINT "Key press ="!!
20 PRINT INKEY$
30 GOTO 10
```

This loop continuously prints the message followed by the value of INKEY$. When no key is being pressed, the value of INKEY$ is a space, otherwise the current key being pressed is displayed. Try pressing any character on the keyboard. As the program passes through line 20, INKEY$ is evaluated as the value of the key being pressed. This is then printed.

As you can see, the program is executing continuously, and does not become stuck at line 10 waiting for a key to be pressed. You will have to use BREAK to stop the program.

If you want the program to wait at the INKEY$, a pause parameter can be specified. The value of the parameter which appears after the INKEY$ is placed in brackets, and specifies the time for which the program will wait for input. If a key is pressed during the pause time, the pause is cut short. The number in brackets is the number of 20 millisecond (one fiftieth of a second) frames which the program will wait at the INKEY$ before continuing anyway. If the number is 0, the program returns immediately, as it does for the default condition (no number specified). If the number is −1, it will wait forever for an input. Type in the following:

```
20 PRINT INKEY$(100)
```

This will make the computer wait for 2 seconds for your input before giving up and continuing with the program. Try pressing some keys and watch the effect.

Now try typing in

```
20 PRINT INKEY$(-1)
```

and watch the difference.

INKEY$ is a special type of identifier. It is the name of a variable which has a clearly defined value in a given situation. It is different from, say, the variable A or NAME$ which can have any value, and must be assigned that value at some time. INKEY$ is an example of a *function*. The parameter

which follows it in brackets is called an *argument*, and it is used by the function to alter its value or action in some clearly defined manner. Here, the argument is the number of 20 millisecond frames for which the program stops and waits for an input. Most functions have at least one argument, and arguments are always put in brackets. Most arguments can be numeric expressions, and do not have to be pure numbers. When the function is evaluated, we say it *returns* a value. Thus we could say that INKEY$ is a function whose argument specifies a pause period, and which returns a string value containing the first character pressed on the keyboard during that pause period. The default value of the argument is zero.

Another function for input from the keyboard is KEYROW. The keyboard on the QL is arranged as a matrix of keys with 8 rows (numbered 0 to 7) and 8 columns (numbered 1,2,4,8,16,32,64,128). The KEYROW function takes the row number as its argument, and returns the column number of any key which is pressed on that row. Its value is zero if no key is being pressed on that row when it is evaluated. The columns and rows are reproduced in the QL manual, in the keyword list, and the following is an example:

```
10 PRINT "Column no.="!KEYROW(5)
20 GOTO 10
```

This will print 0s until a key is pressed on row number 5. Try pressing R then Y then I then O. You will see that 16, 64, 4 and 128 will be printed respectively. These are the column numbers associated with those keys which are all on row number 5. Try pressing several keys at once and you will see that the number returned is the addition of the key numbers. If you understand binary numbers you will realise that the column numbers are powers of 2, and that this allows you to tell which keys are pressed when more than one is held down at any time.

Notice that the rows and columns do not follow the rows and columns as set out on the actual keyboard layout itself.

Clock

The QL has an internal clock which acts just like a digital watch. The clock is available for setting and reading using some BASIC statements and functions. These are SDATE, to set the clock, ADATE, to adjust the clock by a set number of seconds, and DAY$ and DATE$ functions for reading the clock. These last two return the day and date when evaluated..

Try clearing the screen and typing in:

```
PRINT DAY$
PRINT DATE$
```

This will tell you the day and date in the present setting of the clock, and

show you the formats of the DAY$ and DATE$ functions. Just like the functions of the last section, DAY$ and DATE$ are special identifiers with given definitions. They are defined as strings of characters which contain the day as a word, and the current date respectively. Thus the display shows the time at which the function DAY$ and DATE$ were evaluated.

Later in this book we will see how to extract parts of a string. This will allow you to use just part of the DATE$ function in a program.

SDATE is a *statement*, which means that it commands the computer to perform an action. SDATE is used to set the time, and has to be followed by six numeric expressions, normally numbers, between commas, which give the following information:

```
SDATE Year,Month,Day,Hour,Minute,Second
```

An example of SDATE would be:

```
SDATE 1985,8,17,23,1,0
```

This will set the clock to precisely 11.01pm, on 17th August 1985 as soon as the statement is executed. Use the statement to set the clock to the present time, and then print DATE$ to view it. To see the time ticking away, type in:

```
10 PRINT DATE$
20 GOTO 10
```

Running this will show the time changing. Use BREAK to stop the program.

The ADATE statement is used to alter the date by a number of seconds. It is followed by a numeric expression which tells the clock to advance or move back by that number of seconds. If the numeric expression gives a positive number, the clock advances by that number of seconds. If negative, the clock moves back instead. It allows fine adjustments to be made. For instance, if you use

```
ADATE 35
```

then the clock will advance 35 seconds, and continue. If you use

```
ADATE -300
```

then the clock will move back 5 minutes, and continue.

The DATE function is numeric, and allows the date to be read and stored in a compact numeric form, to save memory if you are storing lots of times for some reason.

Try typing:

```
PRINT DATE
```

As you can see, the value of this function is a number. It can be converted back to a readable string by using the DATE$ or DAY$ functions. Both of these functions can take an argument, and if they do, they do not refer to the

internal clock for their value, but use the argument.

For instance, type in the following program, using NEW and CLS if necessary.

```
10 TIME = DATE
20 PRINT "The numeric date is:"!TIME\\
30 REM Now convert into a string
40 TIME$ = DATE$(TIME)
50 PRINT "The normal date form is:"!TIME$
```

Line 40 takes the argument (the variable TIME) and uses it to produce an output which is in the normal DATE$ format. It has not referred to the internal clock. To see this, type in:

```
GOTO 40
```

a few times. You will see that the time displayed remains constant. If you now print DATE$ on its own, you will see the actual time.

The DAY$ function acts in exactly the same way, but will pick out the day information in its argument. Try

```
PRINT DAY$(TIME)
```

This will print the day from the value of the constant variable TIME.

These functions and statements will be used again in examples, and you will find them very useful in programs which are meant to be run according to day or date. Time is a very useful input to a program.

Memory I/O

There is another class of Input/Output, which has to do with the internal memory of the QL itself. This allows you actually to alter and read this internal memory.

As you probably know, the QL has a large amount of internal memory, and this is used to store data and programs, such as the ones which you have tried out above. This memory is organised in a neat and logical manner, and consists of separate boxes. Each box can contain a character such as a number, letter or other symbol. The QL has over 131,000 such memory locations, the contents of which can be written to or read from by the machine. A certain amount of this is taken up by the characters which are displayed on the screen, and some more is taken up by a sort of scratch pad which the QL uses all the time for its normal operations.

The details of the actual memory structure are complex and beyond the scope of this book, and there are two BASIC words, PEEK and POKE, which can be used to read and write to memory locations directly. They are powerful commands, and using them may actually affect the functioning of the machine. Using PEEK and POKE will cause no harm to the machine

itself, but you may lose data or crash the program in some strange way if you are not careful.

Each memory location in the machine has a label or *address* which is often written in a special numbering system called *hexadecimal*. You can see this in the *memory map* of the QL in the QL Manual. Again, this takes us beyond the scope of this book, and you are advised to read a book explaining how computer memory works before you use these facilities.

The statement POKE actually puts any data you wish into a memory location whose address is written as a normal (decimal) number. The general format is

```
POKE address,data
```

This will put the data specified into the address specified. There are also versions of this statement which will gang up two or even four locations at once from the address given, to allow you to input more data with each POKE.

PEEK is a function with an address as an argument. Its general form is:

```
PEEK (address)
```

Its value equals the data stored at that address. Peek also exists in forms which allow two or four locations to be read at once.

Another statement which acts on the basic form of the machine's data is the CALL statement. This is used to execute programs which have been written in the microprocessor's own special language – machine code. This is also a very powerful facility, and you will have to consult a book on machine code to appreciate its use.

Summary

- A PRINT statement may contain a PRINT list of items to be printed, separated by *separators*. The semicolon ensures that members of the list are printed without spaces between, comma tabs to positions 8 spaces apart for each item printed, backslash causes a line-feed whenever used, and exclamation mark is an intelligent space which can be used to separate items by a single space for each ! used. It also prevents items from being split up at the end of a line.
- Each PRINT statement encountered will start its printing on a new line unless there is a separator at the end of the last PRINT list. A PRINT on its own causes a line-feed to occur.
- AT is used to position the cursor for the next PRINT statement.
- CLS clears the screen.
- INPUT is used to input variable values from the keyboard. A message may be printed before input occurs, by placing the message in literals after the word INPUT.

- READ DATA and RESTORE are used to take in values for variables from a block of data stored in the program itself.
- INKEY$ inputs a single character from the keyboard, either instantly, or during an optional wait period. KEYROW returns the position on any given row occupied by the current key pressing.
- The internal real time clock can be set, adjusted and read using SDATE, ADATE and DATE$ respectively.

Chapter Five
Loops and Decisions

We have seen how loops can be set up in a BASIC program using the GOTO statement. However, no proper program-controlled exit method has been described. The loop counter of Chapter 2 was shown being used to terminate the loop at the correct time. This chapter shows you how to set up and exit loops in a number of ways.

One of the greatest powers of the computer is its ability to make a decision, and act upon it. The QL has some very powerful commands for this aspect of computing, and they are described in this chapter to allow you to produce some highly sophisticated software.

You will see that making a decision depends upon the outcome of a logical statement, and so it will be necessary to introduce some logic in this chapter. Again, the QL allows some complex and sophisticated logical statements to be constructed, and you will find a high degree of flexibility in the SuperBASIC language as a result.

Loops

Loops so far have been formed using GOTO statements. For instance:

```
10 PRINT "Looping"
20 GOTO 10
```

This loop is blind, it cannot exit, and will continue until you BREAK it. Even if we include a loop counter, the counter simply counts up, and there is no logic to detect when a particular count is reached, to finish the program.

If you simply want a loop to execute a set number of times, say 10, you might include a loop counter, N, which increments for each pass of the loop, and then, in English, say:

> Execute the loop FOR N = 1, then STEP N by one to the NEXT value, 2, and execute the loop again. Repeat for values of N from 1 TO 10, then stop.

You will be able to understand this statement immediately, as it is written in

plain English, with a few technical words. The equivalent BASIC statement uses the English words FOR, TO, STEP and NEXT to achieve the same thing. This is called the FOR statement. The FOR statement in fact requires two statements. For this example, these are:

```
FOR N = 1 TO 10 STEP 1
(Statements to be executed)
NEXT N
```

The statement starting with FOR is the head of the loop, and the NEXT N statement is at the bottom of the loop. The statements in brackets are within the loop, and the computer decodes this structure as meaning that the bracketed statements are to be executed for values of N from 1 to 10 in increments of 1. In fact, 1 is the default value of the STEP, and if you leave STEP out, the loop counter automatically steps by 1 each time. An example of the use of the FOR statement would be:

```
10 FOR N=1 to 10
20 PRINT N
30 NEXT N
40 PRINT\\"Loop ended."
```

The loop starts at line 10, with the value of N set to 1. Then line 20 is executed for this value of N, and N is printed. The NEXT N is then executed in line 30, where the computer realises that N has not reached 10 yet, and sends execution back up to line 10 where N is incremented by 1 to 2, and the process repeated. This continues until the number 10 has been printed. The NEXT statement then notices that the value of N will exceed 10 if incremented further, and execution passes onto the statement following NEXT. This could be the end of the program, but here there is another PRINT statement to inform you that the loop has finished.

It is not actually necessary to have any statements in the loop at all. For instance, the following:

```
10 FOR I = 1 TO 1000
20 NEXT I
```

could be used to provide a delay in a program. Normally you would use the PAUSE statement, but the PAUSE delay can be terminated early by a keyboard pressing, and this may be undesirable under some circumstances. The whole FOR statement could be placed on the same line using colons:

```
10 FOR I = 1 TO 1000:NEXTI
```

Again, any statements between the FOR statement and the NEXT statement will be executed, but there are none here.

Let us see some simple uses of this looping method. The following prints the first 12 values in the 12-times table.

```
10 FOR N = 1 TO 12
20 PRINT N*12
30 NEXT N
```

This loop executes line number 20 twelve times, printing the numbers as it goes. The display is simple and unattractive. To improve it, the words "First", "Second" etc. could be added to the printout. However, these cannot be generated by computer, and they have to be input to the program verbatim, and printed out as required. They could be put into a DATA statement. Try the following:

```
10 CLS
20 RESTORE
30 REM - Loop starts
40   FOR N=1 TO 12
50   REM - Input next word
60   READ NUM$
70   PRINT NUM$!"Value","="!!12*N
80   REM -  Loop ends here
90   NEXT N
100 REM - Words required:
110 DATA "First","Second","Third"
120 DATA "Fourth","Fifth","Sixth"
130 DATA "Seventh","Eighth","Ninth"
140 DATA "Tenth","Eleventh","Twelfth"
```

The first two statements give the program a clean slate. The REM statements explain the working of the program. You should be able to see that it executes the statements within the loop 12 times. These are lines 50, 60, 70 and 80. See how the loop itself stands out if it is indented a little from the line number. This loop prints the correct word on each line, followed by the value of that line of the table. Just as the data statement is exhausted, the loop counter reaches its limit, and the FOR loop ends. The result of the FOR loop ending is that the statements following the NEXT in line 90 are executed. However, there are no executable statements here, and the program ends naturally.

There are still improvements which can be made to the display, such as adding in the numbers which are being multiplied together in each line.

Try using EDIT 70 to change line 70 to:

```
70 PRINT NUM$!"Value","="!!"12 X"!N!!"="!12*N
```

To analyse this PRINT statement, you must pay attention to every single character in the PRINT list. The first four characters form the identifier for the next word contained in the DATA statement. The next character is a single space separator (!). Then comes a group of 5 characters (Value) to be printed literally. If you look at the DATA table, you will see that the number of characters in the word stored in NUM$, plus a space, plus the 5 literal characters is always greater than 8, but less than 16. This means that the first

two 8-space columns on the screen are used up, and the comma which comes next always starts the = sign at the start of the third column. The next two spaces, followed by the group of four literal characters (12 X), with its space (!) take up the next 7 spaces. N is then printed out with 2 spaces, and this can take up 3 or four spaces depending upon the value of N. The = sign and space, followed by the value of 12*N can, therefore, start in either of two positions on the screen, and accounts for the lack of columnisation of the last part of the display on each line.

This explanation is involved, but only uses principles introduced so far. It is worthwhile halting here for long enough to assimilate this information, and ensure that you understand it. PRINT is one of the most important statements in the language, and you will need to use it in more and more complex ways as you learn more about loops and decisions.

In general, in BASIC, the FOR statement does not need to use integers for the loop counter, it can start at any value, end at any value, and step by any increment. These values can also be positive, negative, or given by general numeric expressions. However, early versions of the QL will not allow you to step the loop counter below zero.

As an example of a non-integer loop counter, the following program evaluates a numeric expression which might be used for some type of scientific application. Statements are also included to show you how fast the program runs. You should use NEW before typing this program in:

```
10 CLS
20 REM - Print the starting time
30 PRINT "Start Time is"!!DATE$\\\
40 N = 0
50   REM - Loop starts here
60   FOR Y = 2.5 TO 0 STEP -.5
70   N = N + 1
80   X=(1-Y)/(10-Y)*8.75E8
90   PRINT "Value no."!!N,"="!X
100  REM - Loop ends here
110  NEXT Y
120 PRINT\\"End time is"!!DATE$
```

This program starts the loop counter at 2.5, and subtracts 0.5 from it with each pass of the loop until the number 0 is reached. The variable N is used to provide the correct number in the message in the PRINT statement. Notice that N should be set to 0 at the start to ensure that it is clear for use in the loop.

To check your machine, try changing the number after the TO from 0 to −1 and see the effect. If the output oscillates, and the loop never ends, then the FOR statement will not allow the loop to go negative.

The FOR statement has some other attributes which will be introduced after the next section on decisions. There are also other SuperBASIC words associated with loop structures which will be introduced later.

Decisions

In the flowcharts of Chapter 2, decisions were introduced to cause the program to execute along different paths depending upon the outcome of the state of a variable, or some other parameter. In English, we often use conditional words such as 'if' and 'then' for this type of situation. We might say, 'If a certain condition is met, then print the value of a variable'.

IF and THEN are perfectly good BASIC words, and they mean exactly what they say. As an example of the use of these words, you will remember that when GOTO was introduced, the loops which we tried simply went on forever until BREAK was typed. Using IF, we can set up this sort of loop, and exit from it when a particular condition is met. A simple form of the IF statement is:

IF condition THEN GOTO line number

When the condition is satisfied, the program goes to the given line number. If the condition is not met, the next statement after the IF statement is executed instead. For instance:

```
10 CLS
20 N = 0
30 PRINT "This is pass number"!N
40 IF N = 10 THEN GOTO 70
50 N = N + 1
60 GOTO 30
70 REM -  Program ends here
```

This is so near to English that you can probably follow the logic with little help. The first statement after the CLS sets up a loop counter, and the next statement PRINTs a message followed by the loop counter value, which at first will be zero. The following statement then checks to see whether N has reached 10 yet, which it clearly has not on the first pass. If the condition were true, then the program would execute the GOTO statement following THEN, and the program would end. The condition is false in the beginning, the statement following THEN is ignored, and the program passes onto line number 50 where the loop counter, N, is incremented. The next line sends execution back to line 30, and the process repeats. After 11 numbers have been printed, the condition between IF and THEN becomes true, and the program goes to line 70 and ends.

It is most important to set the initial value of the loop counter correctly. It may have been a surprise to you to find that this program actually PRINTS 11 numbers, while the IF statement checks for the number 10. The reason is simply that the numbers 0 to 10 are printed, as you can see by running the program. To print the 10 numbers from 1 to 10, you could have set N to 1 in line 20. To print the 10 numbers from 0 to 9 you could simply change the order of the statements in the program. For instance, try swopping the statements in lines 40 and 50 and running the program. This will cause 0 to

be printed as the first number, and the program will end with 9 being printed. This is a very important lesson, and you should pay attention to the starting and ending values of a loop to ensure that it deals with the exact set of numbers required for any application.

The IF statement gives a method of executing a set of instructions depending upon the truth of a given condition. In the examples above, the GOTO statement, after THEN, was executed if the condition were true. Any other statement can be placed after THEN on the same line, even multiple statements separated by colons. They will all be executed if the condition is true. Also, the condition can be considerably more complex than simply checking for a given value.

The flowchart in Fig. 2.5 of Chapter 2 can now be written as a program. It is a good exercise for you to try this for yourself before continuing. The next section describes the process.

The flowchart is set out in a form which will convert to a program with ease. The > sign is used in the same way, after the IF, and this provides the condition for checking. Figure 5.1 shows the process used to swop the values

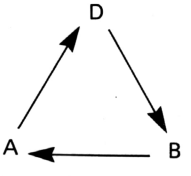


Fig. 5.1. Swopping variable values.

of two variables. The values are shown moving around the sides of a triangle to illustrate the process. To swop the values of A and B, A has to be stored somewhere, in variable D for instance, and then B's value is stored in A. B is then set equal to D. Refer to the flowchart in Fig. 2.5 to see the logic of the following program. The same variables are used, and their values will be input by you when you RUN the program:

```
10 CLS
20 PRINT "Please type in 3 numbers,"
30 PRINT "each followed by ENTER."
40 PRINT
50 INPUT A,B,C
60 REM - Swop A & B if A>B
70 IF A>B THEN D=A:A=B:B=D
80 REM - Swop B & C if B>C
90 IF B>C THEN D=B:B=C:C=D
100 REM - Swop A & B if A>B
110 IF A>B THEN D=A:A=B:B=D
120 PRINT
130 PRINT "Ascending order is:"
140 PRINT A,B,C
```

Lines 70, 90 and 110 perform the swops if necessary, and leave the numbers alone otherwise. The result is that A, B and C contain the numbers in the correct order. If you need to refer to the explanation of the process, turn back to Chapter 2. RUN the program a few times, with different numbers, including negative numbers, to see what happens.

Before exploring the IF statement in further detail, let us see what can be added to the above program to make it run over and over again until you decide to finish. We could do with a method of making the program ask if you wish to run it again, and if not to end. This can be achieved by using the INPUT statement to input the answer to a simple question. Add the following lines to the program and RUN it to see the effect:

```
150 PRINT
160 INPUT "Run again?"!!ANS$
170 IF ANS$="yes" THEN GOTO 10
180 PRINT\\"Thanks, I'm going now."
190 PRINT\\"       GOODBYE!"
```

Line 160 asks you whether you wish to run the program again, and you should type in an answer. The string of characters which you type in will be input into the variable ANS$. Line 170 then checks the contents of this variable, and performs a comparison with the word yes, in lower-case letters. If you did type in yes, the program starts again from the top. If not, it ends with a message. Note that when asked the question, if you answer anything but yes in lower-case letters, the program will end. Try answering with anything else, and see the effect.

If you wish to make the program more intelligent, you could add a line that checks for an upper-case YES after line 170, and also goes back to the start of the program if the answer is true. You could even check to see if you have typed NO or no. If you have typed none of these words, then the program should tell you that you have made a mistake, and ask you to retype your answer. It is an excellent exercise to try and modify the program to perform these tasks. A later example will use these ideas, and you will see then how to achieve it if you do not succeed now.

This sort of interaction between user and machine is essential when writing programs for people who are not interested in programming, but simply want the machine to be as self-explanatory as possible.

BOOLEAN relations

To construct more complex logical conditions after the word IF, we will have to look at ways of combining simple logical conditions such as N=10 and A>B. This will allow any type and shade of condition to be constructed, and hence make the decision more intelligent. This subject is part of a study called 'Boolean Algebra' after George Boole, the nineteenth century mathematician who founded many of the laws of logic which we use today in modern computing.

In Boolean Algebra, statements can have two values: TRUE and FALSE.

Thus:

grass is green = TRUE

while:

2>15 = FALSE

The statements on the left-hand side of these equations are logical entities, and just as for numeric variables, their value (TRUE and FALSE) can be printed and checked. Thus, if we were to invent a computer which did not work on numbers, but rather worked on logical statements instead, we might be able to write:

```
PRINT grass is green
```

As the words are not contained within literals, these words would not be printed literally, but rather the *value* of the expression would be printed as TRUE.

The following statement:

```
PRINT 2>15
```

would print FALSE on the screen. The entities in the PRINT list are called *Boolean expressions* to distinguish them from numeric or string expressions.

The equivalent in a numeric computer would be:

```
PRINT 12/3
```

and the screen would show the number 4 rather than printing out the characters 12/4.

In this logical computer, the logical values of logical expressions would be manipulated just as for numeric values and statements. There would even be some equivalents to +, −, * and so on. Whether or not this type of computer would be of any use is a question which is outside the realm of this book. However, BASIC does allow logical statements to be evaluated as to their truth, so that complex conditions can be checked in statements such as IF THEN.

In SuperBASIC, a logical condition or statement does have a value TRUE or FALSE, as broadly introduced above. However, these values are not given by the words TRUE and FALSE, but by some numeric values which are equivalent. For instance, the computer gives the value 0 to FALSE logical expressions, and any non-zero number to TRUE expressions. Thus, try the following:

```
PRINT 1>2
PRINT 2>1
```

The first PRINT will interpret the expression in the PRINT list as a Boolean expression because of the > sign. It will check its truth or otherwise, notice that it is FALSE, and thus PRINT 0. The second will return a non-zero number as it is TRUE.

You can fool an IF statement by knowing the way in which this works.

For instance, try:

```
IF 0 THEN PRINT "FALSE"
```

This will print nothing because the condition after the IF has a value zero, and is thus interpreted as FALSE. Now try:

```
IF -15 THEN PRINT "TRUE"
```

This will display TRUE on the screen as the IF condition is a non-zero number and is interpreted as TRUE.

To operate decision statements, it is essential to be able to combine conditions, and make more complex ones to fit into the IF statement. For instance, the program in the last section could have benefited from being able to ask questions such as: 'Is ANS$ equal to YES or yes?'. The word *or* here is used to combine the two possibilities. This would have allowed us to check upper- and lower-case versions of the answer in the same statement.

There are several logic operations which can combine and build up logical expressions in SuperBASIC, and they are:

AND, OR, XOR and NOT

These words are called *Boolean operators*, just as +, −, * and / are called *numeric operators*. Generally, these operators have the same meaning in BASIC as they have in English, but they are more precise. The only one which may be unfamiliar is XOR. They are described below.

AND

This is used to combine two conditions which must both be TRUE for the expression to be TRUE. For instance:

```
1>0 AND 2>3 = FALSE
```

because the second expression is not TRUE. However, the following is TRUE:

```
12/4=3 AND 15>3 = TRUE
```

because each of the Boolean expressions has the value TRUE.

Try printing these as shown above to see that they are correct.

OR

This is used to combine two expressions, to give a Boolean expression which is TRUE if either or both of the conditions are TRUE. For instance,

```
1>0 OR 3>2 = TRUE
```

because both conditions are TRUE. Also,

```
0>1 OR 15/5=3 = TRUE
```

because one of the conditions is TRUE. The following is FALSE:

```
0>1 OR 18/3=16 = FALSE
```

because both conditions are FALSE. Again, try printing these expressions to check them.

XOR

This is the exclusive OR operator, and is similar to OR. It gives a TRUE answer if one of the conditions is TRUE. However, the case of both being TRUE is excluded, and gives a FALSE value to XOR. Using the same examples as for OR, compare the following results:

```
1>0 XOR 3>2     = FALSE
0>1 XOR 15/5=3  = TRUE
0>1 XOR 18/3=16 = FALSE
```

This agrees with OR except for the first expression where both parts are TRUE, which is excluded by XOR.

NOT

This is not used to combine two expressions, but can be used to modify the meaning of a logical condition. For instance:

```
NOT 2>3
```

is the same as saying 2 is not greater than 3. See if you can predict the outcome of printing this, and then try it to see if you were right.

We will see these Boolean operators used in examples later.

Relations

The symbol > is called a *relation*, and is one of a set which is used to compare numeric and string variables. There are six relations in the full set, and they are:

Greater than	>
Less than	<
Greater than or equal	>=
Less than or equal	<=
Equal	=
Not equal	<>

They are all used in the same way as the > sign, and they can be used to compare the values of both numeric and string variables.

It is usually possible to write a comparison using these relations in a number of ways. For instance, if A is greater than B, then B is less than A. The expression A>B could, therefore, be written as B<A.

It is also perfectly correct to write it as:

```
A>=B AND NOT A=B
```

You will have to inspect this quite carefully, and think about the second expression if you are unfamiliar with this type of logic. Try to find other expressions which are equivalent to this. You will find that it helps to say the relations to yourself in English.

A program example

This section describes a simple game which will test and improve your reaction time for using numbers on the keyboard. The object of the game is to press a number displayed on the screen within a certain time limit. The program uses INKEY$, and several Boolean expressions.

The program starts by displaying a message to tell you what to do. You have to type in a number which the computer uses to determine how fast the game will run. The numbers allowed are from 0 to 9. The higher the number you choose, the slower it will play. After choosing the speed, another message tells you what to do next. You are told to press any key to play the game. The computer then chooses a random number, and displays it on the screen. If you manage to press that keyboard number in time, the computer tells you that you have won. The longest time you have to press the key is a little under a second. The fastest is probably too fast unless you are very lucky. When you understand the program, you can modify it as you wish.

After you have typed in the program, some instructions are given for you to save the program on a microdrive cartridge for later use.

There is one function used in this example which you have not met before. It is a function for producing random numbers. The function RND(N) will return a random number between 0 and N. The RND function will be described in full in Chapter 7.

The program should be typed in with great care to include every detail of the following list. As you type, you will make typing errors, and have to correct them. This will give some useful experience in using the QL for programs which are too long to fit on the screen. Use the EDIT function on lines which are too long to retype for one single error, and use the LIST function to check parts of the program when you have finished. Remember that you can LIST just a part of the program. Also, if a LIST gets out of hand, and threatens to scroll lines off the screen which you want to inspect, you can always stop a LIST by BREAK. Notice that for compactness a minimum number of spaces have been used.

When you RUN the program, it could contain a number of your typing errors. EDIT lines which the QL tells you are in error, and compare with the listing in this book. Correct as necessary.

```
10 CLS
20 REM *** THE REACTION GAME ***
30 PRINT"    HOW FAST ARE YOU?"
40 PRINT"THIS GAME WILL SHOW YOU"
```

```
50 PRINT
60 PRINT"CHOOSE A SPEED LEVEL 0-9"
70 PRINT"0 IS FAST, 9 IS SLOW"
80 REM
90 REM SPEED LEVEL INPUT NOW
100 REM
110 PRINT:INPUT"SPEED?",SPD%
120 IF SPD%<0 OR SPD%>9 THEN
    PRINT"INPUT A NUMBER FROM
    0 TO 9 PLEASE":GOTO 110
130 CLS
140 PRINT"YOUR SPEED LEVEL IS"!SPD%
150 PRINT
160 PRINT"YOU WILL BE GIVEN"
170 PRINT"      A NUMBER"
180 PRINT"THE GAME IS TO PRESS"
190 PRINT"THAT KEY NUMBER FAST!"
200 PRINT
210 PRINT"PRESS ANY KEY TO PLAY"
220 REM
230 REM NEXT LINE WAITS FOR KEY
240 REM
250 PAUSE -1
260 REM
270 REM RANDOM NO. CHOSEN
280 REM
290 RN%=RND(9)
300 PRINT
310 PRINT"PRESS THE NUMBER:"
320 PRINT:PRINT,RN%,"NOW!!"
330 REM
340 REM WAITS SHORT TIME FOR ANY
350 REM KEY TO BE PRESSED
360 REM
370 KEY$=INKEY$(1+SPD%*5)
380 REM
390 REM YOU LOSE IF NOT A NUMBER
400 REM
410 IF KEY$<"0" OR KEY$>"9" THEN
    GOTO 500
420 VAL=KEY$
430 REM
440 REM CHECK RIGHT KEY PRESS
450 REM
460 IF VAL=RN% THEN GOTO 520
470 REM
480 REM WIN OR LOSE MESSAGE
490 REM
500 PRINT:PRINT"BAD LUCK - YOU LOSE"
```

```
510 GOTO 560
520 PRINT"WELL DONE!!  YOU WIN"
530 REM
540 REM PLAY AGAIN?
550 REM
560 PRINT
570 INPUT"ANOTHER GAME?",ANS$
580 IF ANS$="YES" OR ANS$="yes"
    OR ANS$="Y" OR ANS$="y"
    THEN GOTO 10
590 IF ANS$<>"NO" AND ANS$<>"no"
    THEN GOTO 570
600 PRINT:PRINT"          BYE!!!"
```

The first section of this program is a set of PRINTs to put messages on the screen, and an INPUT to input the speed parameter stored in SPD%. This is an integer variable, and even if a non-integer is input, it will round it off. Line 120 checks to see that the player has input a number between 0 and 9. If not, a message is displayed to warn the player to input a valid number. This repeats until he does. The next section is a message print again, to make the display interactive, and self-explanatory.

Line 250 then uses the PAUSE statement with a number which makes it pause until a key is pressed. When it is, RN% is set equal to a random number between 0 and 9, inclusive. The RND function does not produce an integer, and RN% rounds it off to an integer, as well as storing it. This number is then displayed, and the player tries to press that keyboard number before the pause argument in the INKEY$ function runs out. Whatever value INKEY$ has when it is evaluated is stored in KEY$ for later use. The argument in INKEY$ is an expression made up from SPD%, so that the larger SPD% is, the longer the pause. You can lengthen the pause by multiplying SPD% by a larger number than 5.

If a key is not pressed before the pause runs out, INKEY$ will not return a number; if a key is pressed before the pause runs out, that key's character is stored in KEY$. Line 410 checks to see if the character in KEY$ is between 0 and 9. You will see that exactly the same relations exist between strings as those for numbers. This is explained fully in Chapter 6. If KEY$ is not a number between 0 and 9, then the program goes to line 500, and tells you that you have lost. If it is, the coercion concept is used to change it to a number, rather than a string. Line 420 sets a numeric variable equal to KEY$. The QL automatically changes from a string stored in a string variable, to a number stored in a numeric variable. Line 460 then checks whether the number you pressed was correct, by comparing it with the random number in RN%. If it is, a winning message appears; if not, you lose.

Line 570 is reached either way, and you are asked whether you wish to play again. Line 580 checks for four ways of answering in the affirmative. You can type the word yes or just the letter y in upper- or lower-case. The next line checks that ANS$ does not contain the word no in upper- or lower-

case. If it does, the program ends, if not then the player must have input something wrong, and he is asked to repeat his input.

The IF statements associated with the question and answer between man and machine are constructed to trap any possible input.

This program gives a fair amount of practice of the work so far, and you should try to understand it and use some of the ideas for your own programs.

If you wish to store this program, the following section tells you how to do it. Chapter 9 describes the use of the microdrives in more detail.

Storing the program

If this is the first time you have used or stored a program, you should refer to the QL manual, and observe the precautions there with great care. Remember that the cartridges contain magnetic tape, and the information on them can be wiped by any stray magnetic field. A favourite way to lose data is to bring them near to an object which conceals a magnet – for instance, loudspeakers in transistor radios, or the magnetic paper clip holders one finds on people's desks.

Having read the precautions, if you have a blank tape which has never been used before, you must FORMAT it first. Remove it from its outer plastic cover, and place it in the left-hand microdrive slot. Type in the following exactly:

```
FORMAT MDV1_PROGS1
```

Do not forget the underscore (_) character. This will format the tape for saving programs and data, and call it PROGS1 for later identification. Do not FORMAT a tape which has data on it, as this command will wipe it clean.

The next step is to SAVE the program as a file on the tape. You can call the file by any name you wish, but the following SAVEs the program as GAME:

```
SAVE MDV1_GAME
```

To see that it has been SAVEd, you can inspect the directory of the tape, which keeps a record of all the files on the tape. Type in the following:

```
DIR MDV1_
```

This should return a message concerning a number of sectors, followed by GAME. It is good practice, and may save disasters, to save more than one copy of the file. Take a second tape, and repeat the above procedure. This will give you a main copy, which you should use all the time, and a backup copy which should be checked, but only used if the main copy fails.

To load the program again, type in the following:

```
LOAD MDV1_GAME
```

When the cursor returns to the screen, you can RUN it even if the tape is still running. However, do not remove the tape until it has stopped.

SuperBASIC facilities

The majority of the facilities we have seen so far have been standard to most dialects of BASIC. The rest of this chapter and much of the rest of the book describe facilities which are more characteristic of the QL and SuperBASIC. This is not to say that other computers do not have these facilities, but they are less common.

The rest of this chapter describes the ways in which loops and decisions are normally written in SuperBASIC, and we shall see that it promotes the art of writing what are known as *structured programs*.

'Structured programming' is the practice of setting a program out in a conventional and easily readable form. This would entail, for instance, writing the main activities in a program as a set of easily modified modules. One advantage is that the program conforms to a known and agreed type of structure, and this helps to make it *portable*, or transferable to other languages, and to other people. If you take up the habit of setting out your programs in a consistently structured manner, they will be neat, easy to read, easy to transfer to other machines, and easy to modify at a later date. These are important advantages, but sometimes the actual programs produced are longer than would be the case if they were simply written as an unstructured block of computer code.

The REPeat statement

The REP statement is the simplest of the SuperBASIC structured features. It replaces the GOTO statement in loop structures. It uses a *label*, or *loop identifier* to determine the limits of the loop rather than a line number. The normal rules of identifiers apply to this label. In a REP statement, the start of the loop would be of the form:

```
REP loop1
```

loop1 is the loop identifier, and other SuperBASIC keywords act on this loop identifier to repeat or exit the loop. EXIT is, in fact, a SuperBASIC keyword, and when followed by a loop identifier will command the program to leave that loop. It will often be used in an IF statement. The REP statement has two forms. These are referred to as 'long' and 'short'.

The short REP loop allows a complete loop structure to be contained on a single line by using colons to separate the statements. The following shows a typical short REP loop:

```
10 N=0
20 REP LOOP1:N=N+1:PRINT N:
   IF N=10THEN EXIT LOOP1
```

(Notice that when you type this in, the QL turns REP into REPeat.)

If you examine this multiple statement, you see that there are three statements before the IF statement, including the REP. Then comes the IF statement itself, and it contains a fifth statement, which is the EXIT command. Everything on line 20 is considered to be within the loop identified as LOOP1. These statements will execute over and over again until an EXIT statement is met containing the correct loop identifier. This only happens when N has counted up to 10. At that point, the logical relation in the IF statement is evaluated to TRUE, and the EXIT statement following THEN takes the program out of LOOP1, and the program ends.

The EXIT only ties up with its REP statement through the loop identifier LOOP1. If this is left off the EXIT, the computer tells you there is an error when you try to write the line. If you put another label there instead, the computer will accept it into the program, but gives an error when the EXIT is executed. Also, you can still use a GOTO statement after THEN to finish the loop if you wish. This is less common in this type of structure, as the use of EXIT with the loop identifier is also a good way of causing the program to be self-documenting in some sense.

A further point concerns the IF statement. Any statements written after THEN are considered to be part of the IF statement, and not part of the loop. Thus if you move the PRINT statement from its present position to following the EXIT statement, using a colon, it will only be executed when N reaches 10. Try it and see. Use EDIT to change this line; do not retype it completely for the change.

Line number 10 is essential, as RUN does not reset the state of the variables, and if N is set to some value above 9, the loop will never end. Make sure you understand why this is the case.

Statements can be placed before the REP on line 20, and they will not be included in the loop. The REP is the head of LOOP1, wherever it occurs. Try changing the program to:

```
10 N=0:PRINT"START":REP LOOP1:N=N+1:
   PRINT N:IF N=10THEN EXIT LOOP1
```

The loop now starts in the third statement of line 20, and no other lines are necessary. This type of compaction is useful for small routines such as this which perform a small closed activity. This simple program element would take up 6 lines of program if fully expanded. However, it is not always a good idea to compact everything for the sake of space savings, and a program written in this manner can be confusing and difficult to read and correct. You will have to find a balance by experience.

The long REPeat form contains statements which are spread over several lines, and the end of the loop is identified using END REP followed by the

loop identifier. The loop can still be terminated by EXIT as above. The following is the same example again, but written using END REP:

```
10 N=0
20 REP LOOP1
30 N=N+1:PRINT N
40 IF N=10THEN EXIT LOOP1
50 END REP LOOP1
```

This allows a set of program lines to be looped until EXIT is executed. It allows more sophisticated constructions to be formed, in a readable way, using simple loops.

The REP philosophy is to use labels for loop statements, rather than having to use line numbers. This allows a meaningful word to be used for a given routine, instead of just a number. It also saves having to work out line numbers while writing the program, and it helps you to see the extent of the loop as you are working. If you are used to other BASICs, you will quickly see the advantage of these labelled structures.

The following program is self-documenting; read it carefully before continuing.

```
10 CLS
20 REP INPUT_NUMBER
30 INPUT"INPUT A NO.
   FROM 100 TO 200",X
40 IF X>=100 AND X<=200
   THEN EXIT INPUT_NUMBER
50 PRINT"FOLLOW THE INSTR'NS!"
60 GOTO 30
70 END REP INPUT_NUMBER
80 PRINT"YOUR NUMBER IS:"!X
```

The loop called INPUT_NUMBER is repeated until the user inputs a valid number. Note that you cannot use a space within an identifier, but the underscore character is allowed for this exact purpose. The identifier is meant to be meaningful, and to give some indication as to what is happening.

The loop starts at line 20, and includes all the statements down to line 70, which is at the bottom of the loop. When the correct input is found in the IF statement, the EXIT is executed, and the program continues at the line following the END statement for the loop. Try running the program.

NEXT can also be used with the REP loop, and it has a similar effect to END REP. It can be used in an IF statement to force the next pass of the loop to be taken. The following program illustrates the point:

```
10 CLS:N=0
20 REP LOOP
30 N=N+1
40 IF N<5 THEN PRINT"N<5":NEXT LOOP
50 PRINT
```

```
60 PRINT "N>=5"
70 IF N=10 THEN EXIT LOOP
80 END REP LOOP
```

The loop starts at line 20 with N at 0. N is then incremented, and found to be less than 5, so THEN is executed, followed by the PRINT, and NEXT LOOP. This causes the program to go back to the start of the loop in line 20. When this loop yields a value for N which is greater than or equal to 5, then lines 50 to 80 are executed for each pass of the loop. This continues until N reaches 10 and the loop finishes at the END REP statement. An EXIT, therefore, leaves the loop at the END statement, while NEXT forces the present pass of the loop to be cut short, and the next pass started. This can be very useful, as this example shows, for executing part of a loop under one condition, and another part under another condition.

STOP, CONTINUE, RETRY

There is another way to end a loop using the STOP statement. This is similar to using BREAK, but is written into the program. When the program encounters a STOP, it ceases execution, and returns to the command mode. This can be used in a number of ways to help to debug a program. Also, you can CONTINUE after a STOP, and the program starts from the next line. Try adding STOP in a new line number 55 in the last but one program. When you enter a number out of the required range, the program tells you to follow the instructions, and then stops. You can CONTINUE here, and the program will continue as if nothing had happened. If you change anything in the program, then you cannot normally continue in this way. Remove line 55 again to make the program run normally.

RETRY is used to continue from where an error occurred, by retrying the last line again to see the error occur again. This can help you to debug the error. To see this, RUN the program, and when it asks for numbers, type in some letters of the alphabet. On pressing ENTER, the program comes out into command mode, and tells you that there is an error in the expression in line 30. Now type RETRY, and the program will redo that line. This time input numbers, and the program will run correctly.

Extensions to the FOR statement

SuperBASIC also allows a short compact form of the FOR statement, along with a long structured form using an END FOR statement in line with the REP statement.

The short form of the FOR statement is of the following form:

```
FOR X=I TO J STEP K: statements
```

Notice that it does not require a NEXT statement. It will simply execute the

statements after the colon until the loop variable X runs out. This is a very convenient way of compacting a simple loop down to one line. Again, this should not be used as a substitute for writing out an involved FOR loop as separate lines, but is useful to perform a separate and complete task of a simple nature. For instance, you might want a particular printout surrounded by asterisks. This would be a good way to do it:

```
10 CLS
20 FOR I=1 TO 22:PRINT"*";
30 PRINT
40 PRINT"* THIS IS A PRINTOUT *"
50 FOR I=1 TO 22:PRINT"*";
```

This type of display handling can be useful to make otherwise dry printouts look more interesting. Also, it helps to highlight different areas, if you do not want to use colour or graphics.

The short form of the FOR statement is useful for the simple jobs which loops perform, while the long form, described next, allows some of the most complex loop structures to be produced.

As we saw earlier, the FOR statement uses a NEXT to define the bottom of the FOR loop. We can also use the keywords END FOR, for exactly the same job.

For instance, the following two programs have identical effects:

```
10 FOR I=1 TO 50
20 PRINT I
30 NEXT I

10 FOR I=1 TO 50
20 PRINT I
30 END FOR I
```

The difference between NEXT and END FOR comes in the use of EXIT. EXIT was introduced above in the REP statement. You should remember that when EXIT was executed, the program immediately went to the next statement after the END REP statement. This is the key to the difference between END FOR and NEXT. If there is a NEXT and an END FOR statement in a FOR loop, EXIT will cause the next statement after the END FOR to be executed.

The following program shows how this can be used:

```
10 CLS
20 FOR COUNT=1 TO 5
30 INPUT"DO YOU WISH TO STOP?",ANS$
40 IF ANS$="YES"THEN EXIT COUNT
50 NEXT COUNT
60 PRINT"COUNT NOT STOPPED"
70 END FOR COUNT
80 PRINT "PROGRAM ENDS"
```

The FOR loop has two endings, a NEXT and an END FOR. The EXIT in line 40 only uses the END FOR. If you answer with YES in line 30 then the IF statement condition is TRUE, and EXIT COUNT is executed. This causes line 80 to be executed, and the message PROGRAM ENDS appears. If you now answer with anything but YES, line 50 executes, and the program jumps back to line 30. If you continue answering in this way, the variable COUNT reaches 5, and the statement after the NEXT is executed.

You should remember that EXIT causes the statement after END FOR to execute, and finishing the loop counting causes the statement after NEXT to execute. These facts will help you to work out how any combination of the statements works.

Try swopping lines 50 and 70. This time, when you answer YES, and the EXIT in line 40 is encountered, the statement after the END FOR is in line 60. Thus you will see that message appear. The NEXT statement is then encountered in line 70, and the loop variable is incremented, and line 30 executed again. This process then repeats until either the count of 5 is reached or you answer with anything but YES. If the count finishes, the statement following NEXT in line 70 is executed, and the program ends with the ending message. If you do not answer YES, then the statement following END FOR which is now in line 50 is executed. The message in line 60 is printed, and the NEXT in line 70 continues the count. There is no way to exit the program until the count is complete.

Normally, NEXT comes before END FOR, and the block of statements between is referred to as a *loop epilogue*. It is only executed if the loop is not EXITed, and comes to a natural end as the loop counter runs out.

Nested loops

All the loop structures on the QL are allowed to be placed inside each other, and this is called *nesting*. An example of two nested FOR loops is:

```
10 CLS
20 FOR OUTER=1 TO 3
30 PRINT"    ***    OUTER    ***"
40   FOR INNER=1 TO 5
50   PRINT "INNER LOOP"
60   NEXT INNER
70 NEXT OUTER
```

Examine the FOR NEXT pairs which define the two loops. The outer loop executes 3 times, and each of these passes causes the inner loop to execute 5 times. This is indicated on the printout. The outer loop starts with the variable OUTER equal to 1. Line 30 then displays that the outer loop has started, and the program enters the inner loop at line number 40. This FOR loop uses INNER as its variable, and as it steps up, the words INNER

LOOP are printed on the screen. When INNER comes to the end of its count, the statement after NEXT INNER is executed, and the outer loop is re-entered. The process thus repeats until both loops are exhausted.

You can nest further loops within loops inside the inner loop. The most important rule to observe is that you must always fully enclose one loop inside another when nesting. Do not allow any of the statements in one loop to fall outside the other so that one loop crosses the boundary of the other. For instance, it is not permitted to swop lines 60 and 70 above.

It is quite normal for further statements to be placed between the inner and the outer loops. Try adding in a PRINT statement between lines 60 and 70, and this will be printed each time the program falls out of the inner loop.

The IF and END IF

The IF statement introduced so far has only been used in its short form, whereby if the condition is true, all the statements after THEN are executed as long as they are on the same line, separated by colons. The IF statement is also available in two long forms. One of these uses END to allow several lines of program to be executed from THEN to END IF. The general form of this statement is:

```
IF condition THEN
statements
END IF
```

The first statement sets up the condition which, if TRUE, will cause all the statements between THEN and END IF to be executed. If FALSE, the program continues immediately after the END IF.

For instance, as an example of this type of statement, let us look at the problem of causing one or two routines to be executed depending upon some outcome. The following program either clears the screen, and puts a message in its centre, or lists itself on the screen and informs you that this is the current program. It also checks that you have followed the instructions correctly, and loops until you do. Notice how REP is used to replace GOTO. In fact, it is possible to do without GOTO entirely from now on.

```
10 REP THIS_PROGRAM
20 INPUT"type y or n",A$
30 IF A$="y" OR A$="Y" THEN
40 CLS
50 AT 5,10: PRINT "THIS PROGRAM
   DEMONSTRATES END IF"
60 EXIT THIS_PROGRAM
70 END IF
80 IF A$="n" OR A$="N" THEN
90 CLS:LIST
```

```
100 AT 0,0:PRINT"THIS IS THE
     CURRENT PROGRAM"
110 EXIT THIS_PROGRAM
120 END IF
130 PRINT"YOU HAVE NOT FOLLOWED
    THE INSTR'NS"
140 END REP THIS_PROGRAM
150 REM THIS IS THE END OF THE
    PROGRAM
```

This is an example of nesting. We say that the IF statements are nested within a loop, formed by the REP and END REP statements. To read this program correctly, look for the REP and END REP statements, to see the extent of the main loop. Then find the IF and END IF statements, to give you some idea of the modular structure of the program within the loop. When you have located these, analyse each one separately. This is the normal approach for reading structured programs.

Everything between line 30 and line 70 will be executed as being after THEN if the condition is TRUE. The same is so for the lines between 80 and 120 for the second IF structure. If neither of these THEN blocks are executed, it means that the user of the program has input an invalid letter, and the program loops back and repeats until one of the THEN blocks is executed. Each block of statements after THEN contains an EXIT. Try the program, with both valid and invalid key pressings to see the effect.

The ELSE statement

The IF statement has one other long form which uses the keyword ELSE. It gives the IF statement more power to choose from various possibilities. The full structure is of the form:

```
IF condition THEN
statements
ELSE
statements
END IF
```

As the English equivalent of this implies, the statements between THEN and ELSE are executed if the condition is TRUE, and then program execution goes to the line following the END IF. If the condition is FALSE, the statements following the ELSE are executed instead, followed by the lines following the END IF.

The following gives an example of this structure:

```
10 REP LETTER_CHECK
20 INPUT"TYPE THE LETTER K",A$
30 IF A$="K"THEN
40 CLS
```

```
50 PRINT"CORRECT LETTER"
60 EXIT LETTER_CHECK
70 ELSE
80 PRINT"WRONG LETTER"
90 END IF
100 END REP LETTER_CHECK
```

Again, when you see the REP at the head of the program, look for the END REP statement to see the extent of the loop. Then view the IF statement, and look for END IF and/or ELSE. This will help you to appreciate the modules which are written in the program, and hence the structure.

If the condition after IF is TRUE, then lines 40 to 60 are executed, which ends the loop. If the condition is FALSE, the statements following the ELSE are executed, and these end at the END IF, which does not finish the loop. The loop is then executed again and again until the correct letter is typed in.

You should take a break at this point, and try an exercise. See if you can rewrite the reaction game program, introduced earlier, in a new form using the more structured approach described above. In particular, try to remove all the GOTO statements, and replace them with REP statements.

Nested IF statements

The use of IF statements can be extended further by nesting. This gives some of the most complex structures in SuperBASIC and an example is given below, which you will have to follow through carefully:

```
10 CLS
20 PRINT"INPUT 3 DIFFERENT NOS."
30 INPUT"EACH WITH ENTER",A,B,C
40 IF A>B THEN
50   IF B>C THEN
60   PRINT
70   PRINT "DESCENDING ORDER"
80   ELSE
90   PRINT"ORDER MIXED UP"
100  END IF
110 ELSE
120  IF C>B THEN
130  PRINT "ASCENDING ORDER"
140  ELSE
150  PRINT"ORDER MIXED UP"
160 END IF
```

The indentations and lines above are meant to help you to follow the logic of this program. When you RUN this program, follow the instructions exactly; do not enter two or three numbers the same. However, do try

numbers in all sorts of different orders.

The IF statement in line 40 has its ELSE in line 110, and its END in line 160. It has two separate IF statements nested within it. The first of these checks to see if B>C given that A>B from the outer IF statement. If B>C, then you should convince yourself that the numbers must be in descending order. If not, then the order is mixed up.

If B>A, then the THEN in line 40 is not executed, and the program jumps to the ELSE in line 110. If C>B, then the numbers are in ascending order. If not, they are mixed up.

Note that as for FOR statements, programs will not work properly unless nesting is complete, and there is no interlacing. Do not allow part of one IF block to cross another. You can nest IF statements as deep as you like, as long as each is fully contained in the next outer.

It is often better, from the program writing point of view, to check all the possibilities of a given set of conditions in separate IF statements, rather than using the nested structure above. This will help to make the program readable, and better structured. However, some very compact and neat software can be written by nesting IF statements. You should pay careful attention to the exact way in which the ELSE and END IF statements tie up with their IF statements. It is easy for the program and the programmer to become confused!

Selection loops

The last set of keywords in this chapter are the ones concerning the execution of one of a set of program blocks depending upon one of several outcomes of a decision. The normal BASIC words for this statement are ON GOTO. The QL supports these keywords for compatibility with other languages, but has others which are more powerful and structured. These are SELect, and the typical END SELect statements. The first section below describes ON GOTO.

ON GOTO

When GOTO is executed, it switches the processing to a specified line number. ON GOTO allows the switch to occur to one of several line numbers, depending upon the value of a numeric expression. The general form of the statement is:

ON numeric expression GOTO a,b,c,...

The list of numbers after the GOTO contains line numbers separated by commas. The numeric expression must take on a value between 1 and the number of line numbers appearing in the list after the GOTO. The value of

the numeric expression gives the placing in the list, from left to right, of that line number chosen for execution. For instance:

```
ON 3 GOTO 400,50,55,4000,500
```

would execute line number 55. This is rather a long way of saying GOTO 55, but it illustrates the point. A more practical example might be:

```
ON N GOTO 20,50,100,500,2000
```

This is a method of executing one of a number of routines dealing with different aspects of the value of N. This statement may be in a loop, or it may be a method of switching to one of 5 routines from a special display called a *menu* which asks the user to press a number depending upon which function he would like to select next. We will see an example of this in the next section.

It is important that the numeric expression lies within the valid limits, otherwise an error is given. It does not have to be an integer, as it will be rounded off correctly.

SELect and END SELect

The more structured method of performing the task of the last section is to use the SEL statement. In common with the other SuperBASIC structures, there is a short and a long form, and the latter uses END SEL to finish the function.

The short form is as follows:

```
ON X = A TO B: statements
```

The statements after the colon are only performed if X is in the range A to B inclusive.

The short form is very useful for replacing a rather cumbersome form of IF statement. For instance, in a previous example, we had to check whether a variable input by the user was within a certain range. If the variable was X, and its valid range was between 100 and 200 inclusive, we had to write:

```
IF 100<=X and X<=200 THEN statements
```

This can be written more easily using the SEL statement as:

```
SEL ON X=100 TO 200: statements
```

The statements, all on the same line separated by colons, are executed if X is within the inclusive range 100 to 200. The values of X are rounded off to integers, and this means that, for instance, 99.5 and 200.49 are considered to be in the range. Similarly, the computer rounds off non-integer limits. For instance, the following is identical to the above:

```
SEL ON X=99.6 TO 200.39: statements
```

You should really only use this function for integers.

The long form of SEL is used to execute one of several blocks depending upon the value of a variable, as for ON GOTO. In this case, the first line of the SEL block is of the form:

SEL ON variable

This sets up the variable to be used in subsequent selection lines of the form:

ON variable = A TO B
statements
ON variable = C TO D
statements
ON variable = REMAINDER
statements
END SEL

This constitutes a select list. The block of statements after the first ON statement is executed only if the variable is in the inclusive range given by A and B, which may be numeric expressions. If this statement block is executed, the program then goes straight to the line following the END SEL. Each statement block is terminated either by the next ON statement, or by the END SEL. The last member of the select list above uses the optional keyword REMAINDER. If none of the previous conditions is satisfied it supplies another block of statements to execute if required.

An example of this type of structure would be in allowing the user to input a number to select a particular item from a menu on the screen. As a simple case of this, consider a program which prints one of three names depending upon which number is chosen from the menu. Even though the following program is a little long, it performs this neatly, and in an easily read structure using SEL.

```
10 CLS
20 REP PROG
30 REM - PRINT OUT MENU
40 PRINT"CHOOSE A NUMBER AS FOLLOWS"
50 PRINT\'(1) PRINT "MARY"'
60 PRINT\'(2) PRINT "GEORGE"'
70 PRINT\'(3) PRINT "JOHN"'
80 PRINT\"(99) EXIT PROGRAM"
90 PRINT
100 REM - INPUT YOUR MENU CHOICE
110 INPUT"NUMBER?",CHOICE
120 PRINT:PRINT
130 REM - SELECT PRINT OR EXIT
140 SEL ON CHOICE
150 ON CHOICE=1
160 PRINT"NAME CHOSEN IS MARY"
170 ON CHOICE=2
```

```
180 PRINT"NAME CHOSEN IS GEORGE"
190 ON CHOICE=3
200 PRINT"NAME CHOSEN IS JOHN"
210 ON CHOICE=99
220 PRINT"THANKYOU - BYE BYE!"
230 REM - PROGRAM ENDS HERE
240 EXIT PROG
250 REM - TRAP BAD INPUT
260 ON CHOICE=REMAINDER
270 PRINT"CHOOSE AGAIN!!"
280 GOTO 110
290 END SEL
300 REM - SCROLL DISPLAY UP
310 PRINT\\\\
320 END REP PROG
```

Lines 50 to 70 use single quotes as literals because double quotes are contained in the message.

Notice that this program gives an example of a long SEL statement nested within a REP statement. Nesting statements in this manner is sufficiently natural that it requires no special thought. Notice also that there is a GOTO statement in line 280; it is a good exercise to try to replace this by a REP structure. The REM statements above take you through the program, which you should examine with great care.

You can see how the use of the REMAINDER simplifies the problem of trapping incorrect keyboard entries. Compare this with the same problem in previous examples which used IF statements. Imagine, also, how you might perform the function of this program using IF statements. You would learn a lot by rewriting this program using IF and GOTO to perform the same actions. If you do, you should try to structure the program by writing each module to be performed as a separate module at the end of the program, and selected by IF and GOTO as required.

Summary

- FOR STEP...NEXT is used to give loops of a set length. END FOR can be used to terminate the FOR loop when an EXIT is encountered. A short FOR may be placed on a single line, followed by a multiple statement. No NEXT is required.
- IF THEN is used to execute a block of statements depending upon the truth of a logical statement. A block of statements can be terminated by an END IF. ELSE allows a block to be executed if the IF condition is false.
- Boolean statements are logical statements, whose value may be TRUE, given by a non-zero value, or FALSE, given by 0. AND, OR, NOT and XOR operators can be used to build up complex Boolean conditions.

- REP is used to introduce loops ending in END REP, which may be exited using EXIT.
- STOP CONTINUE and RETRY are used to cease execution, and then continue either with the next statement, or by repeating the last one executed.
- Placing loops and IF statements within each other is called *nesting*. No nested entity may overlap the outer entity.
- SEL, END SEL and REMAINDER are used to select one of a number of alternative statement blocks depending upon the value of a given variable.

Chapter Six
Strings and Arrays

This chapter deals with two concepts which have already been introduced – *strings* and *arrays*. Strings can be manipulated using special functions and statements, and these are described in this chapter. An array was given as an example in Chapter 2, and this chapter defines the way in which subscripted numbers are defined and manipulated in SuperBASIC.

This chapter also introduces a special numeric code which is used for characters, and shows how to compare strings using the same logical relations as are used for numbers.

Ordering and storage of strings

Everything in a computer is stored numerically. We have seen that even the truth of a Boolean statement is stored numerically, i.e., TRUE is stored as a non-zero number, and FALSE as zero. The same is true of the characters which make up the keys on the keyboard. Each character is stored as a number according to a standard international code called the *ASCII Code*. This stands for American Standard Code for Information Interchange. Thus a string is stored as a sequence of numbers, each one uniquely corresponding to one of the characters in the string. This is one of the reasons why you have to identify a memory location by an identifier which automatically defines whether or not it is a string. Otherwise, the numbers corresponding to strings would become confused with ordinary numbers – they all look the same both to us and to the computer.

The ASCII code list is given in the QL manual, and you should refer to this for the following.

To see how the numbering of the characters works, there are two functions which allow you to view the actual code number which corresponds to each character. These functions are called CODE and CHR$. CODE takes in a string as its argument, and returns the ASCII code corresponding to the first character in the string. For instance, type in:

```
PRINT CODE("A")
```

This will return the ASCII code number 65. Now try:

```
PRINT CODE("ABCD")
```

This will give the same answer, as only the code of the first character of the string is printed. To see the codes of the others, type in:

```
PRINT CODE("B"),CODE("C"),CODE("D")
```

This will show you that these characters have codes which are in ascending order. To convert a numeric code to its corresponding character, you can use the CHR$ function. The $ sign in this identifier signifies that the function has a string value. Try typing in:

```
PRINT CHR$(65)
```

and you will see the letter A printed, to confirm that number 65 is the ASCII code for the upper-case A. You can use this function to discover the ASCII code for each character. You should use CLS then type in the following program, and inspect the result carefully.

```
FOR I=32 TO 127:PRINT CHR$(I);
```

It simply prints the complete set of characters which your QL can display. As you can see, the code numbers run from 32 to 127. Other code numbers produce non-printing characters which are used by the system for other purposes. You should be able to see that there is a space at the start of the printed set; this is the first character in the set, and has ASCII code 32. The rest of the characters begin with exclamation mark. They then run through some other symbols and the digits from 0 to 9. Some further symbols come next, followed by the upper-case letters. More symbols are followed by the lower-case letters, and finally the last few symbols. The set ends at number 127 with the copyright sign, which is non-standard and special to the QL.

In most BASICs this numerical code is used to order the string characters to allow strings to be sorted using the relations:

```
>     >=     <     <=
```

Normally, if one character has a lower ASCII code than a second character, the first is defined to be less than the second using the < relation. This type of comparison is used as the basis for the function of sorting lists of words into alphabetical order. Just as in a telephone directory, words are normally sorted by first letter, then by second letter and so on. However, this causes confusion if the ASCII code is used, because, if you look at the character set on the screen, the lower-case letters are all later in the set than the upper-case letters. This means that a list sorted according to ASCII code would put the word apple after Zebra, because the ASCII code for Z is lower than that of a. The QL solves this problem by defining the ordering of the string characters quite differently from the ASCII code. If you use any of the standard relations, the actual order of letters is as follows:

AaBbCcDdEeFfGgHh etc.

This puts the situation right, and allows proper alphabetic sorting regardless of case. If you want to sort according to ASCII code, you can use the CODE function to convert to ASCII, perform the sort, and then convert back.

The actual order of the characters according to this new method of comparison is incorrectly stated in the early version of the QL manual. We shall see, shortly, from an example program, the correctly ordered set and how it compares with the ASCII ordering.

This explains how four of the standard relations work on strings, but there are three more string relations, and they are:

```
=      <>      ==
```

The first is the normal equals, and two strings must be identical in every way, including case, to satisfy this relation. The second is not equal, and any difference will satisfy this relation, including a difference of case. The third is called *equivalent* or *almost equal*. Two strings, for instance, are equivalent if they have the same letters and numbers, but one or more of the letters are of different case. The following statements give some examples. To test examples of your own, simply print the relation. A zero answer means it is FALSE, a non-zero answer means TRUE. Refer back to Boolean relations if you are unsure of this point. The following are examples of these relations:

```
"ABC"    =   "aBC"    is FALSE
"Fred"   ==  "FRED"   is TRUE
"Fred"   ==  "Freda"  is FALSE
"FRED"   <>  "fred"   is TRUE
```

In order to handle strings of letters, it is important to be able to store them in a numerically sequenced manner. This was introduced in Chapter 2, and you should now refer back to remind yourself of the section which dealt with an array. Arrays for both strings and numbers are introduced next.

Arrays

An array is a set of variables indexed by numbers, normally from 1 or 0 up to a maximum number depending upon how many values have to be stored.

Suppose you wish to store 100 numeric values in a set of array variables. You must first choose an identifier. We shall use VALUE for this illustration. This identifier can only be used for one value as it stands, and we attach a number from 1 to 100 to it to index the full 100 values we wish to store. This is attached in BASIC by using round brackets. Thus the 100 variables which are needed, one for each value to be stored, would be called:

```
VALUE(1), VALUE(2), . . . , VALUE(100)
```

The numbers in the brackets can be replaced by a numeric expression or a variable, and each one of the items in the above list is treated just as for any normal identifier such as X or COUNT. For instance, PRINT VALUE(65) will cause the 65th value to be printed.

In order to use any array, space has to be reserved for it in the computer's memory. Most dialects of BASIC have a statement to perform this task. It means that you have to know how many values you are going to store from the beginning, or at least a maximum which you might need. You should not use too high a number, however, as it will use up too much memory space, and might restrict your programming space.

The DIM (or dimension) statement is used to reserve space for an array. Its general form is:

DIM identifier (number of elements)

This statement defines the top index which the identified array can have starting at 0. For instance:

DIM VALUE(100)

allows the array VALUE to have elements:

VALUE(0), VALUE(1), . . . , VALUE(100)

This is actually 101 elements, but it is often convenient to forget the 0-indexed element. This depends upon your application.

If you forget to use DIM in a program, an error will be caused unless you restrict yourself to indices from 0 to 10. However, it is good practice to use DIM every time, and we shall do so in this book. An error is also caused if the array index exceeds that specified by a DIM statement.

We can now see how to fill an array with values, and how to print it out using a loop.

The following illustrates two of the many methods of setting an array to a set of values:

```
10 DIM VALUE(100)
20 FOR I=1 TO 100
30 VALUE(I)=I*I
40 NEXT I
```

```
10 DIM VALUE(100)
20 FOR I=1 TO 100
30 INPUT"TYPE A NUMBER",VALUE(I)
40 IF VALUE(I) = -1 THEN EXIT I
50 END FOR I
```

The first of these uses assignment to fill the array. The second takes in numbers from the keyboard, and allows the loop to be stopped by typing in −1, if you do not want to fill the whole array. A FOR statement can now be used to print the array elements.

```
FOR I=1 to 100:PRINT VALUE(I);
```

This can only be done without a DIM statement, because you have just run a program which dimensions the array. The DIM statement also erases the array. Try using DIM VALUE(100), and then try printing the array as above; you will find that all the elements are set to 0.

Arrays can have more than one subscript, and the DIM statement gives the upper value for each subscript. The following shows an example of a 3-dimensional array:

```
10 DIM A(12,34,20)
20 FOR DIM1=1 TO 12
30 FOR DIM2=1 TO 34
40 FOR DIM3=1 TO 20
50 A(DIM1,DIM2,DIM3)
   =DIM1+DIM2+DIM3
60 NEXT DIM3
70 NEXT DIM2
80 NEXT DIM1
```

The first statement dimensions the array to have indices of up to 12 in the first dimension, 34 in the second and 20 in the third. The program then uses 3 nested FOR loops to fill the array with values each of which is the sum of the indices of that element. You will find that nested loops are a very common structure for the handling of arrays.

The DIMN function can be used to find the values of the dimensions placed in the DIM statement. It could be useful for a program to check that it is not going to exceed the allowed dimension value. DIMN has the following form:

```
DIM (array identifier , dimension)
```

For instance, if the program wished to check on the value of the second dimension in array A above, it would evaluate:

```
DIM(A,2)
```

and the answer would be 34. If the dimension number is left out in the DIMN function, the default is the first dimension. Thus the following would return the number 12:

```
DIM(A)
```

Strings can be stored in arrays whose identifiers end in a $. The only difference is in the DIM statement. The DIM statement contains an extra number to tell the computer the maximum number of characters which will be stored in any of the string elements of the array. This extra number is always the last number of the array. DIMN acts on string DIM statements exactly as for numeric arrays. The last number, giving the string length, is also accessible using DIMN, but you should check your machine, as you may find that it adds 1 to the correct value for values between 1 and 9.

The following fills a string array with 20 elements, and each of these array variables can have up to 100 characters.

```
10 DIM NAMADD$(20,100)
20 FOR K=1 TO 20
30 PRINT"INPUT NAME AND ADDRESS"
40 INPUT"100 CHARACTERS MAX:"!NAMADD$(K)
50 NEXT K
```

This program first dimensions a string array called NAMADD$, so that it can have 20 elements, each with up to 100 characters. Then it executes a FOR loop to input 20 names and addresses. If you input more than 100 characters, the program will simply store the first 100, and cut off any others. Inputting any less will cause the string variable to be packed out with spaces after the input characters.

As an example of the ideas in this chapter so far, and to practise some of the concepts from previous chapters, we can now return to the problem of ordering strings using comparisons.

An example of string ordering

This gives an excellent illustration of the use of nested FOR loops and arrays, as well as a general program for performing the *bubble sort* which was mentioned in Chapter 2. You should refer back to the flowchart on sorting numbers into ascending order in Chapter 2. We will now do this for the 96 ASCII codes, and sort them according to the QL type of ordering.

To see how this works, recall that the first part of the 3-number sort routine simply bubbled the largest number up to the right-hand side of the set. In other words, if the numbers were:

6 1 3

The first step was to swop 6 with 1, and then with 3 to produce:

1 3 6

The next step was to leave the 6 alone, as it has already taken up its correct position, and work on the others, performing the same operation again. This time, they do not need swopping, and the process is complete. If you have more than 3 numbers, the process is identical, but takes more passes. The first job is to bubble the largest number in the set up to the top. For instance, observe the following 6 numbers:

1, 6, 2, 34, 0, 4

In the first pass, 1 and 6 are not swopped because 1 is below 6. 6 and 2 are swopped, and the numbers become:

1, 2, 6, 34, 0, 4

Then 6 and 34 are not swopped, but 34 and 0 are, and then 34 and 4. This gives:

1, 2, 6, 0, 4, 34

34 can now be forgotten as it has arrived at its final destination. The left-hand 5 numbers are now treated in the same way, and their largest number, 6, bubbles to just below 34. Then 4 is already there, and stays put, and so on. There are two loops going on here. One of them keeps count of the size of the left-hand set of numbers, and the other loops within that diminishing set, bubbling the largest number up to the top. The following program will sort all 96 characters into ascending order according to the > comparison.

```
10 CLS:DIM A$(96,1)
20 REM - PUT CHARACTERS INTO A$
30 FOR I=1 TO 96:A$(I)=CHR$(I+31)
40 REM - OUTER LOOP
50 FOR COUNT=1 TO 96
60 REM - INNER LOOP
70 FOR SWOP=1 TO 96-COUNT
80 IF A$(SWOP)>A$(SWOP+1) THEN
90 REM - SWOP CHARACTERS
100 D$=A$(SWOP)
110 A$(SWOP)=A$(SWOP+1)
120 A$(SWOP+1)=D$
130 END IF
140 END FOR SWOP
150 END FOR COUNT
```

The first line clears the screen and dimensions the array A$ to have 96 elements of 1 character each. Line 30 then stores the 96 characters in A$ using a short FOR loop, and the CHR$ function with argument from 32 to 127. The outer loop then starts, and counts up from 1 to 96 as it is executed. For the first of these counts, COUNT equals 1, and the inner FOR loop counts from 1 to 95 in line 70. For each of these 95 counts, the bubble sort works to move the highest character up to the top position by swopping as necessary. For this value of COUNT, which is 1, the last comparison in line 80 is performed on A$(95) and A$(96). This shows that the full set of 96 characters, stored in A$(1) to A$(96), is covered. The inner loop finishes in line 140 when the final comparison and possibly a swop has been performed. The outer loop then counts up to 2, and only A$(1) to A$(95) are involved in the inner loop this time. The process continues until the rearrangement is complete.

Do not be alarmed at the time it takes to run the program; it has 96 elements to bubble sort. The first pass of the outer loop causes the inner loop to look at all 96 characters, and bubbles the highest to the top. Then the inner loop looks at the remaining 95, then 94 and so on. This gives a total of:

$96 + 95 + 94 + \ldots + 1$

inner loop passes. See if you can type in a FOR loop for immediate execution which will give you the answer to this addition, and tell you how many loops are performed. You will have to start by setting a variable equal to zero, and using that variable within the FOR loop for the addition.

The set of program lines given below will print the 96 characters in two ways. First, it prints the set of character codes in the special order produced by the program above. Second, it prints the same set in ASCII code order so that you can see the difference. If you have already run the program above, then the array A$ will be set up. Just type in the following lines, use CLS, and type RUN 160.

```
160 REM - NOW PRINT THE TWO SETS
170 PRINT "COMPARISON ORDER IS"
180 PRINT
190 FOR I=1 TO 96:PRINT A(I);
200 PRINT:PRINT
210 PRINT"ASCII CODE ORDER IS:"
220 PRINT
230 FOR I=32 TO 127:PRINT CHR$(I);
```

You will see that as far as > is concerned, *full-stop* is the lowest order character, while in the ASCII code *space* is at the bottom of the set. You can also see that the numbers are below all the letters in both sets, and that they are in their correct numerical order. Notice how the upper- and lower-case letters interleave, in the comparison set, as one would expect for a telephone directory.

String manipulations

To be able to extract or replace part of a string, we use the string slicing statements of SuperBASIC. This will allow you, for instance, to extract just the month from DATE$. So far this has been locked up within the string, and has only been available for printing along with everything else in DATE$. Another application would be to make operator input more intelligent. When asked for YES or NO from the keyboard, you could just extract the first letter of the response, and as long as that is correct, the program could proceed without having to ask for a more accurate response to the question.

'String slicing', as it is called in SuperBASIC, is performed in a very different way in other BASICs, and anyone used to other BASICs will see the advantage of the QL's method fairly quickly.

The allocation of strings in a computer system can be treated in two distinct ways. Many machines simply allow a fixed length of storage to be set aside each time the computer comes upon a string identifier in a program. This is either wasteful of space, particularly when using arrays, or

is insufficient for really long strings. The QL, however, allows variable length strings. When the program comes across a string assignment, it reserves enough space for that assignment to be honoured. If a longer assignment comes along later, the size is increased accordingly. The space allocation does not reduce when it meets a shorter assignment.

This variation in memory allocation is not effective for string array variables, and that is why you have to specify the length of each string in the array by the last number of a DIM statement. Thus all members of a given array have the same length of memory allocation.

The memory allocation for a string is important for string slicing as we shall see.

To describe string slicing we will take the following example of a string, and show how to extract and change parts of it. The string example chosen is given by the following assignment which you should type in after using NEW and CLS:

```
TEST$ = "HELLOFREDSMITH"
```

This reserves 14 places for the string TEST$, and stores the letters shown there. The places in the string are considered to be numbered from left to right. You can extract some of these by specifying a range in brackets after the string name. For instance, try the following:

```
PRINT TEST$(6 TO 9)
```

This will extract the 6th to the 9th characters in TEST$, and print them. The string TEST$ is unaffected by this action, as you can see by printing it. You can make up parts, or *substrings*, or TEST$ using this slicing technique. For instance, try:

```
T1$=TEST$(1 TO 5)
T2$=TEST$(6 to 9)
T3$=TEST$(10 TO 14)
```

Then type the following:

```
PRINT T1$!T2$!T3$
```

This will show that T1$, T2$ and T3$ have been set equal to substrings of TEST$. Also, the current memory size allocation has now been set for these three strings.

To extract a single character from the string, use a single number of the slice. For instance:

```
PRINT TEST$(6)
```

will give the letter F.

Parts of TEST$ can also be replaced by slicing, using an assignment as follows:

```
TEST$(6 TO 9)="JOHN"
```

Try this and then print TEST$ to see the effect. This will have replaced FRED with JOHN. You now have complete freedom over the contents and manipulation of a string, except for one important consideration – its memory allocation. You have to be careful not to replace part of the string with more than the whole string can take. Just remember that the latest assignment for the whole string determines its present length. Replacements to parts of the string do not affect this memory allocation, and it is up to you to ensure that replacements do not overflow. For instance, TEST$ has 14 elements; no assignment should refer to an element number 15 or more.

If you wish to set up a string of a certain length for later replacement slicing, you should start by filling it with spaces. This can be done as follows:

```
STRING$ = "              "
```

However, this is difficult for long strings. You could use a FOR statement to fill the string N$ with 50 spaces:

```
N$=" ":FOR I=1 to 49:N$=N$&" "
```

However, there is a function called FILL$ which allows a string to be filled with any number of characters. It has the form:

```
FILL$( "A" , X )
```

or

```
FILL$( "AB" , X )
```

The first returns a string of length X, filled with As. The second returns a string of length X, filled with repetitions of AB. If X is odd, the last character is A.

This function can be used to print lines of characters on the screen. For instance:

```
PRINT FILL$("*",30)
```

puts a line of 30 asterisks on the screen.

To reserve 50 spaces in N$ for later slice replacements, use:

```
N$ = FILL$(" ",50)
```

To read just the month in DATE$, print it first and see where the month comes:

```
PRINT DATE$
```

DATE$ is defined in such a manner that the first four characters give the year, then comes a space, then 3 characters for month, then space and so on. The size of DATE$, and the positions of the information stored in it are constant to allow you to slice it predictably. However, you should recall that DATE$ can take an argument in order to convert a numerical date into a string. It is not possible, therefore, to use the slicing parameters with

DATE$ itself, as this would be confused with a numeric date. DATE$ must be read into another string first, and then sliced. This is done as follows:

```
10 CLS
20 PRINT"THE PRESENT MONTH IS:"!!
30 K$=DATE$
40 PRINT K$(6 TO 8)
```

This can also be used to time a particular routine by extracting the minutes and seconds before and after the routine's execution. The following gives you such a display:

```
10 CLS
20 K$=DATE$
30 PRINT"MINS AND SECS:"!K$(16 TO 20)
40 FOR I= 1 TO 10000:N=1
50 PRINT
60 K$=DATE$
70 PRINT"MINS AND SECS:"!K$(16 TO 20)
```

The time is short, and hence is not very accurate, but try adding this to the 96-character bubble sort above, and it will be quite accurate.

Coercion

In order to use time in a program, it will often be necessary to be able to treat the numbers in the DATE$ string numerically, instead of purely as characters. This is where the QL's concept of *coercion* comes in. You can convert a numerical string into a number by a simple assignment. Suppose that the string "12345" is stored in K$. Then in the following statement:

```
K = K$
```

the QL notices that there is a mismatch across the equals sign. The right-hand side is a string, and the left-hand side is a floating point variable (no $ sign). It will try its best to perform what is intended, and so it looks into the string and finds that it is purely numeric. It is therefore possible to convert it to a number, which is stored in K. This is a SuperBASIC facility, and other BASICs normally have special conversion functions instead.

The coercion concept also works the other way round. For instance:

```
K$ = 8504
```

will store the number in the string as a set of characters. You can slice this string to see that it has worked. Try:

```
PRINT K$(2 TO 3)
```

This will produce 50. If you try slicing a floating point number, an error will be reported.

You have to be careful using long numbers in this way. For instance, try:

```
K$ = 273648602
```

Work out the following before trying it:

```
PRINT K$(2 to 5)
```

You might expect 7364. Now try it. The answer is in fact .736. To see why this happens, try printing K$. Before the coercion is performed, any numeric expressions on the right-hand side of the assignment are evaluated. This includes rounding off and converting to E form if necessary. Try printing out various substrings of K$ to gain a feel for the situation.

Coercion will also work for evaluating string versions of numeric expressions. For instance, if N$="10" and M$="20" then:

```
PRINT N$*M$
```

will give 200. Whenever SuperBASIC sees a numerical operation which has string variables instead of numerical variables, it will try to convert in order to complete the operation. It will not always be able to do this, however, particularly if one of the strings contains letters.

Other string functions

There are two more keywords which can be used to manipulate strings. They are LEN which returns the length of a string, and INSTR which determines whether a given string is contained in another string.

LEN is used as follows:

```
LEN(K$)
```

returns the length of the string K$, and could be used to ensure that a given slicing operation is valid before it is done.

INSTR is an operator, and acts between two strings in the following way:

```
A$ INSTR B$
```

This will return 0, which can be interpreted as FALSE, if A$ is not contained in B$. If it is, a non-zero number is returned which is the position of the first character of A$ in B$. This can also be interpreted as TRUE. For instance:

```
K = "FRED" INSTR "ALFRED"
```

would set K to 3, and the following:

```
K = "FREED" INSTR "ALFRED"
```

would return 0 to K.

An application of this would be, for instance, to see how many times a particular town appears in a list of addresses held in an array. Suppose the

100 element array ADD$ holds these addresses and we are looking for the number of occurrences of the town Leeds, the following program would provide the answer:

```
100 COUNT=0
110 FOR I=1 TO 100
120 IF "Leeds"==ADD$(I)
    THEN COUNT=COUNT+1
130 NEXT I
140 PRINT"Leeds appears"!COUNT!"times"
```

This is assumed to be part of a larger program from which ADD$ comes, and hence it does not have a DIM statement, as it would wipe ADD$ clean.

This routine gives an example of an important use of the relation ==. It does not matter, for this program, whether or not the town's name starts with a capital L, as long as it is not misspelled. It is not easy to design a computer program to perform the intelligent activity of guessing the correct town from a misspelled version. The program would have to know all the towns in Britain, or have some other method of viewing the name intelligently.

Another use of the == relation would be in checking operator response to yes or no questions. Previously we had to check through a complicated set of specific comparisons to recognise YES, yes, or anything in between. This can be done immediately using ==.

Summary

- The ASCII numerical code is used to store characters. CODE is used to find the code of a character, and CHR$ takes a code, and returns the corresponding character.
- The order relations such as > do not use the ASCII ordering; they use an ordering more useful for general alphabetic sorting, regardless of case.
- An array is a set of subscripted variables. Array variable identifiers must end in $ for strings, and % for integers. The DIM statement is used to set aside the correct memory space for an array. DIMN is used to read that dimension back if required.
- A string is assigned a memory allocation equal to its length. This may be increased by assigning a larger string value to it.
- A string may be sliced to copy or replace a substring.
- Coercion ensures that an assignment between a string variable and a numeric variable will succeed if possible. This can be used to turn a number stored in a string into a numeric value.
- FILL$ is used to produce a string containing a repetitive character or pair of characters.
- LEN gives the string's length, and INSTR operates to find a substring, along with its start position within a main string.

Chapter Seven
Calculation and Standard Functions

The QL has a large range of scientific functions for complex calculations. The complete use of these functions requires an understanding of mathematics which is beyond the scope of this book. If you do not need these mathematical formulae, you should skip this chapter for the present. You can always refer back to the chapter at a later date.

It is assumed below that you are generally familiar with the mathematics of the functions and operations involved. The chapter is designed to show you how to use the SuperBASIC functions and operators, and aims to give you enough examples to use them accurately.

The early QL manuals contained many errors and this chapter provides you with corrected versions of the statements in the manuals without necessarily referring to those mistakes specifically. You are advised to use the proven examples below as a guide, and try out the functions and operators for yourself before relying on what the manual sets out.

Calculations in SuperBASIC

Calculations in BASIC are normally performed by using expressions containing operators and functions. We have met the main numeric operators of addition, subtraction, multiplication and division, and some of the functions which SuperBASIC includes. The next few sections introduce all the others.

As we have seen, there are two types of numeric variable – *integer* and *floating point*. You should remember that an integer variable rounds off any number which it is assigned to, and stores it as an exact number. One problem is that the range of such numbers is comparatively small. Floating point variables can contain larger numbers, but they do not necessarily store integers exactly. For instance, a floating point calculation may result in the number 2.999999 as its answer, and as far as that calculation is concerned, this is the same as the number 3. However, if your program is checking for 3, it might include a statement such as:

```
IF X=3 THEN etc.
```

The condition in such a statement as this will not be satisfied by X having a value 2.999999.

You must bear in mind that it is not a good idea to rely on a floating point calculation to give an exact answer. A useful arithmetic form of the double equal sign (==) is introduced in the last section to allow you some flexibility in this respect, but you should nevertheless be careful of this situation.

Calculations between integers do yield exact numbers, but they round off to even greater inaccuracies. For instance, the following calculations all give 5 as the same answer:

```
X% = 3.99 + 1.15
X% = 3.99 + 1.45
X% = 10.33/2
```

Using the % sign on the identifier causes the computer to round off to ensure that an integer is stored in X%.

General numeric functions

In general, the form of these functions is:

```
Identifier(argument)
```

where 'argument' can be a numeric expression itself containing functions in general. In fact, in most places where a number is required in a function, operator or statement, SuperBASIC allows a general numeric expression.

The rules of evaluation of arithmetic expressions using the four simple arithmetic operators have already been given. The overriding rule is that brackets, including those around the argument of a function, are evaluated first. Then comes evaluation of the function, followed by the arithmetic operators. Examples of this are given below.

The functions and operators which follow are only useful if you understand the mathematics behind them. As such they are clearly defined here, but little explanation is given as to their meaning. Use the following for reference, and to correct and back up the QL manual where necessary.

SQRT

This is the normal square root function, and must have an argument greater than or equal to zero. The square root of a number, X, is a number which when multiplied by itself gives X. All non-zero positive numbers have two square roots – of equal magnitude, but opposite sign. The QL gives just the positive square root of a number, and it is used in the following manner:

```
PRINT SQRT(4)
```

This will give the number 2. Any numeric expression may be placed in the brackets following the function's identifier. For example:

```
SQRT(SQRT(27*3))
```

would return the number 3. The contents of the inner bracket would be evaluated first, to yield 81. Then would come the inner function SQRT which would return 9. The out SQRT gives 3.

All the functions below satisfy the same rules, and you can enclose function within function as you wish.

INT

This function returns the *next lower* integer to the value of its argument, unless the argument is an integer, in which case it returns the integer unchanged. For instance:

```
PRINT INT(4.9)    gives 4
PRINT INT(4.2)    gives 4
PRINT INT(4)      gives 4
PRINT INT(-4.9)   gives -5
PRINT INT(-4.2)   gives -5
PRINT INT(-4)     gives -4
```

Notice that *next lower* means nearer to minus infinity, and not necessarily nearer to zero.

DIV

This is the integer equivalent of / in floating point. It is an operator, and is used between two integer numeric expressions, and must be followed by a space, unlike /. This operator should not be used between floating point values. The definition of the operator between two integer valued expressions X and Y is as follows:

```
X DIV Y = INT(X/Y)   where X and Y are integers
```

You should not use floating point expressions for X and Y, as they may not actually contain integers.

Examples of the use of DIV are:

```
PRINT 3 DIV 2 gives     INT(1.5) = 1
PRINT 3 DIV -2 gives    INT(3/-2)= -2
```

ABS

This returns the absolute value of a floating point expression, i.e. it returns the same number if positive, and replaces − with + if the number is negative.

```
PRINT ABS(33.156)   gives 33.156
PRINT ABS(-125.7)   gives 125.7
```

MOD

This is an operator, and only acts on integers. As for DIV, it must not act between floating point expressions. It gives the remainder on integer division. The definition is as follows:

X MOD Y = X − (X DIV Y)*Y

This can further be expanded to:

X MOD Y =X − (INT(X/Y))*Y

For instance:

```
PRINT 5 MOD 3      gives 2
PRINT 5 MOD −3     gives −1
PRINT −5 MOD 3     gives 1
PRINT −5 MOD −3    gives −2
```

Notice how the position of the negative number in the arguments of this operator is important.

PI

This keyword will return the value of pi to 7 significant figures. Thus:

PRINT PI gives 3.141593

DEG

This function converts the value of a numeric expression from radians into degrees. For instance:

```
PRINT DEG(2*PI)       gives 360
PRINT DEG(3.141593)   gives 180
PRINT DEG(8.772)      gives 502.5986
PRINT DEG(−3*PI)      gives −540
```

RAD

This is the opposite of the last function, and converts degrees into radians to 7 significant figures. For example:

PRINT RAD(360) gives 6.283185 which equals 2 pi.

SIN COS TAN and COT

These are the usual trigonometrical functions, and are evaluated to 7 significant figures. All the arguments are in radians, and thus degrees must be changed to radians before using the trig functions. The range of values for the arguments is quite large, being −6000 to +6000 for SIN and COS, and −3000 to +3000 for TAN and COT.

When using the constant function PI, you will find that due to the inaccuracy in its evaluation, the functions will not give exact answers. For

instance, try the following:

```
PRINT TAN(PI/2)
PRINT SIN(PI)
```

The first will give a large finite answer, and the second a small but non-zero value.

ATAN and ACOT

These are the inverse trigonometrical functions. ArcTANgent and ArcCOTangent. There is no formal limit to the arguments of these functions. The argument is not, of course, an angle as stated in the QL manual, and is not expressed in degrees, radians or any other dimension. For instance:

```
PRINT DEG(ATAN(1))   gives 45 degrees.
```

EXP

This is the exponential function which gives powers of the exponential constant *e*. Its range is −500 to +500. For example:

```
PRINT EXP(1)   gives 2.718282
PRINT EXP(0)   gives 1
```

LOG10 and LN

These are the usual logarithm functions, to base 10 and base *e*, respectively. Their arguments must be positive and non-zero. For instance:

```
PRINT LN(EXP(13.89))   gives 13.89
```

RND and RANDOMISE

These functions give pseudo-random numbers, which can either be started at the same number each time, or can be used to give lists with unpredictable starts. RND is the function which returns the random number, and RANDOMISE can be used to set its seed. RND can be used with up to 2 arguments to specify the range of random numbers produced.

If RND has no argument, it will return a *floating point* pseudo-random number in the range from 0 to 1, but not including either 0 or 1.

To produce a number within a range, a single argument will produce an *integer* number from 0 to that number inclusive. Thus:

```
RND(13)
```

will produce an integer in the range 0 to 13 inclusive.

A second argument, after the word TO, will give a general inclusive range for the integer. Thus:

```
RND(13 TO 135)
```

will return an integer from 13 to 135 inclusive.

RANDOMISE is used to seed the random number generator in a predictable manner. If RANDOMISE is executed before RND, then the same number or list of numbers will be produced by the use of RND. For instance, the following program will always give the same pair of numbers on the screen:

```
10 RANDOMISE
20 PRINT RND(9)!RND(9)
30 RANDOMISE
40 PRINT RND(9)!RND(9)
50 GOTO 10
```

The first statement starts the random number generator from an internally fixed number which never varies. The next statement takes the first two numbers from the random number generator, and prints them. Line 30 then repeats the reseeding of the generator, and the next statements thus receive exactly the same random numbers as before.

A different seed may be input to the generator, by an argument following RANDOMISE. Thus:

```
RANDOMISE(12)
```

will set the seed to 12. This may be useful if two separate routines are using blocks of random numbers within the same range. Using RANDOMISE with different arguments will ensure that these blocks are different but repeatable. If RANDOMISE is not used in this situation, the blocks of numbers will certainly be different, but not repeatable.

Operators and relations

We have met a number of operators and relations for the numeric and logical functions of the QL, and this section introduces the rest of the available set.

The QL allows bitwise binary operators to be used between decimal numbers. SuperBASIC converts the decimal numbers into their binary equivalents, acts upon them using the specified operator, and then converts back again.

There is also an operator for raising to a power, and a relation for comparing numbers which are nearly equal.

Binary operators

The four binary number (or bitwise) operators available on the QL do not use the equivalent logical keywords; they use the following symbols:

```
AND &&
OR  ||
```

```
XOR ^^
NOT ~~
```

They are used in the same way as the logical operators, as can be seen by the following examples:

```
PRINT 125 && 100   gives 100
PRINT 125|| 100    gives 125
```

Further numeric symbols

The precision with which the QL stores floating point numbers means that even slight inaccuracies in evaluation will cause two numbers to be unequal in the strictest sense. If a routine requires that two numbers be regarded as equal if they agree to the accuracy of 1 part in 10 to the 7th, then the == relation can be used. For instance:

```
PRINT 1.1354761 == 1.1354762
```

will give a non-zero answer because the relation is TRUE. However, the following gives 0 to show that it is FALSE:

```
PRINT 1.1354761 = 1.1354762
```

This is very useful in evaluating infinite series to this accuracy.

Another numeric operation which is available is *raising to a power*. This is provided by the symbol, ^. Thus:

```
PRINT   3^2     gives 9
PRINT 3.2^1.9   gives 9.115594
```

In the rules for the order of evaluation of operators, the ^ sign has a higher priority than multiplication and division. Thus the following expression:

```
5*2^4/10
```

would be evaluated in the following order:

```
2^4 = 16
5*16 = 80
80/10 = 8
```

There are two other numeric functions mentioned in the QL manuals, but they may not work on earlier machines. They are *integer raise to a power*, and the *sign function* (SGN). Integer raise to a power has the following form:

```
A(^)B
```

and acts on integer values only.

The SGN function should return −1 if the argument is negative, 0 if zero, and +1 if positive. Its form is:

SGN(A)

This completes the description of the numeric functions which are available on the QL. The next chapter will deal with the graphics and screen handling facilities of SuperBASIC.

Summary

- SQRT returns square root.
- INT, DIV, MOD act on integers to give next lower integer, integer division and remainder upon integer division respectively.
- ABS gives the absolute value of a numeric expression.
- PI gives the value of pi.
- DEG and RAD changes radians to degrees and degrees to radians respectively.
- SIN, COS, TAN and COT are the trigonometric functions acting on radians.
- ATAN and ACOT are the inverse trigonometric functions.
- EXP, LN and LOG10 give exponentiation, natural logs and logs to the base 10 respectively.
- RND and RANDOMISE are used to produce random numbers.
- AND, OR, XOR and NOT are available for binary numbers.
- ^ raises one variable to the power of another.

Chapter Eight
Graphics and Sound

This chapter is concerned with the QL's ability to display high resolution colour graphics. In other words, it can show general patterns of lines and curves to a high degree of accuracy. This can be appreciated easily by running the Psion software packages which come with the computer. In particular, the program EASEL uses the screen to its utmost, and you should try this program out before programming the graphics yourself. EASEL wil give you a good idea of the capabilities of the machine, as well as supplying some useful ideas for your own programs.

Graphics

The word *graphics* applies to general screen display. So far we have only used the simple graphics of letters, numbers and special symbols. We have not tried to adjust the colours of the display, or to print general patterns on the screen. If you have tried out the Psion software packages on the machine, you may be wondering how the remarkable drawing effects were produced. All these effects are accessible from SuperBASIC.

The exact nature of the graphics which you can display using the QL depends upon whether you have a TV or a video monitor. Some of the more striking effects can only be appreciated on a monitor, where the resolution is much higher.

The screen is divided into separate dots or *picture cells*. These are called *pixels* for short. By using SuperBASIC you can draw to each pixel separately, and even pick its colour.

In order to pick an individual pixel, we have to have a consistent labelling, or numbering system, for the pixels. We use rectangular co-ordinates to specify a pixel, and this will be described shortly.

There are two modes of display possible, called *high resolution mode*, and *low resolution mode*. In high resolution mode, very fine lines and points can be plotted on the screen, but the colour capability is restricted. In low resolution mode, lines and points are thicker, but there are many more colours. The mode you choose depends upon the particular application.

Finally, you do not have to draw geometric patterns by specifying each pixel separately. The QL has many special drawing facilities to help you create the patterns by specifying the numbers in a graphics statement.

The QL's screen

To see the two different formats for printing on the display, simply select the wrong function key after resetting the machine. If you are using a TV for output, reset the machine and instead of selecting TV when asked, press F1.

As we saw in Chapter 1, the result is three windows on your TV. The white one to the left is for listings, the red one to the right is for print output, and the one at the bottom is the command line where input from the keyboard is displayed. Holding down a letter key will show the small version of the QL's PRINT output mode. This display mode is called 'high resolution' as the display can show smaller patterns, and hence cram more information on it. However, as you can see, if you do not have a monitor, the characters disappear off the edge of the screen when you are in monitor mode.

If you have a colour monitor, you should try pressing F2 after RESET, and see the resolution normally used by TVs.

If you have a TV, it is possible to use high resolution mode without losing the frames off the edge. There is a special command which you can use to enter either mode. It is the MODE command, as we shall see.

Press RESET, and return to your correct choice for F1 or F2.

The full screen is divided into 512 pixels across by 256 pixels down. These pixels join together on the screen to produce the effect of a continuous sheet of paper. In fact, the technical QL term for the screen background is PAPER. The printing on this paper is called INK, and the place in which the printing occurs is called a WINDOW. We shall see that a WINDOW can also have a coloured BORDER. A TV will lose some of the pixels off the bottom and right-hand edges, effectively reducing the total number which can be displayed.

When you switch on, and press F1 or F2, the 512×256 pixels are being used in a specific way depending upon whether you have a TV or a monitor. However, you should bear in mind that the display is still showing 512×256 pixels, and the software is simply using it in a special way. You have complete control over the display's characteristics in a program, and we shall see how to use those characteristics during this chapter.

If you have a TV, type in:

```
MODE 4
```

or

```
MODE 512
```

(They are equivalent.)

This will change the mode to high resolution, and reduce the number of colours available from 8 to 4. The reason for this is that high resolution graphics uses a large amount of memory. If you boost one of the characteristics, you lose part of the other to compensate, as the memory available is fixed. Type in some letters, and you will see that they appear the same size as when you tried pressing F1 a minute ago to see the monitor display after RESET. This time, however, the start of the printing is adjusted to ensure that no characters are lost off the edges. This is achieved by reducing the number of characters allowed on the line for the normal printing activities. As you can see, the high resolution mode is not very satisfactory with a television set, and you will have to adjust the TV controls, including the tuning, to produce the best effect. TVs are not good at displaying fine detail.

Now type in:

```
MODE 8
```

or

```
MODE 256
```

and the display returns to normal for a TV. You can change modes in your programs if you wish.

The screen is divided into a grid, and each pixel is numbered according to its distance, measured in pixels, from the left-hand side of the output window, and from its top margin. The pixel numbered 0,0 is the one which is zero from the left, and zero from the top of the window, respectively. Similarly, pixel number 52,100 is 52 pixels from the left and 100 from the top. The next section shows you how to use the BLOCK function to draw individual pixels, as well as blocks of pixels. Remember that on a TV the output window takes up the whole screen, and on a monitor, in the normal mode, it only covers the right-hand side of the screen.

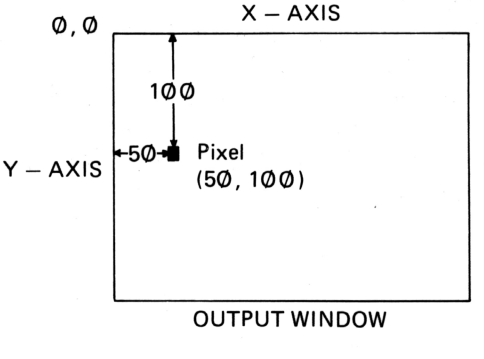


Fig. 8.1. Pixel drawing system.

BLOCK and COLOUR

The BLOCK function illustrates all the features of the pixel display system, and will be used to introduce these concepts.

Figure 8.1 illustrates the output window with the pixel labelling system. This method of labelling points is called a co-ordinate system, and the pairs of numbers shown in Fig. 8.1 are called co-ordinates. Conventionally, the horizontal axis, from left to right is called the X-axis, and the vertical axis is called the Y-axis.

The pixel shown is a small block of colour spaced 50 pixels from the left, and 100 from the top. This pixel is thus labelled with co-ordinates *(50,100)*. The first number is called the X-co-ordinate, and the second is called the Y-co-ordinate. The brackets are not used in SuperBASIC, but are conventional for writing the label down.

The BLOCK function is used to change the colour of a pixel, and the pixel will only show if it has a different colour from the PAPER colour against which it is displayed. The normal screen paper colour on switch on for a TV is white, and if the pixel shown is coloured any other colour, it will show up.

BLOCK is a SuperBASIC statement which takes a number of arguments to tell the computer where to place a rectangular block of one or more pixels, and with what colour. Before seeing the BLOCK function at work, we must define the way in which colour is specified on the QL.

Colour requires up to three parameters for its complete definition, though it can be specified by a composite number, as described in the QL manual. We shall start by dealing with solid colouring, and move to stipple effects later. For the moment, the following table can be used to specify the solid colour of a pixel:

Number	MODE 4 (High res)	MODE 8 (Low res)
0	Black	Black
1	Black	Blue
2	Red	Red
3	Red	Magenta
4	Green	Green
5	Green	Cyan
6	White	Yellow
7	White	White

As you can see, the higher resolution mode has fewer colours, and there are two possible numbers for each colour.

Now we can use the BLOCK statement. The general form of the statement is as follows:

```
BLOCK width,height,horiz,vert,colour
```

This statement draws a rectangle on the screen with the first two parameters specifying the width and height in pixels. The third and fourth specify the co-ordinates of the top left-hand corner of the rectangle, and then comes the colour according to the table above. Try typing the following:

```
BLOCK 100,5,8,20,4
```

This will draw a rectangle of green pixels 100 wide, 5 high and with top left-hand corner at (8,20) in the output window. As a useful exercise, take a rough measurement of this rectangle, and then change the mode and retype the BLOCK command as above, to see that the size of the rectangle stays the same. Then reset the machine, and change to monitor if you are using a TV, and vice versa otherwise. Repeat the above, even if you cannot see it on the command line. Again, the rectangle is the same size, and in the same position in the output window. This makes it clear that no matter what mode is used, the width of a pixel is constant – you will need to remember this for the following.

Reset the machine, and return it to its normal mode. The following program displays all the colours available on your TV or monitor:

```
10 FOR I=0 TO 7
20 BLOCK 20,20,10+I*30,50,I
30 NEXT I
```

The rectangles displayed are of height and width 20, and their colour changes from 0 to 7. Remember that 7 is white, and may not show up against the background. You should now use MODE to change to the other resolution mode, and RUN the program again. This will produce the same rectangles, but will show how the colour changes with the value of the parameter.

The display of individual pixels can only be achieved in high resolution mode, and this does not work well on a TV. However, if you adjust your set carefully, you may be able to see one by changing to MODE 4, and typing in the following:

```
BLOCK 1,1,200,100,4
```

It will be green, and will be in the centre of the window. This is a block 1 pixel wide, and 1 pixel high. If you now move to MODE 8, which is low resolution, you can still specify single pixels vertically, but the horizontal resolution decreases, and only pairs of pixels can be displayed. Specifying a width of 2 or 3 will cause the same 2-pixel wide block to appear. Similarly, a width of 0 or 1 will cause no pixels to appear. To see that the vertical resolution remains the same in either mode, try the following in mode 8:

```
BLOCK 2,1,200,100,4
```

This pair of pixels should show up on most TVs if properly tuned and

adjusted. It will be in the same place as last time, and will be twice the size horizontally. If you now try:

```
BLOCK 2,2,250,100,4
```

you should be able to see the difference between these two blocks, depending upon your television. The second block is twice as high as the first.

Due to the command line at the bottom, and the fact that TVs will not display as much information at the edge of the screen, you will not be able to display the full 512×256 pixels on a TV. The full width of the screen is available on a monitor.

Note that 0,0 in the first two parameters of BLOCK will give a zero-sized rectangle, and in low resolution mode it is the same as specifying 1,0 in this place.

COLOUR

The other parameters of colour allow you to change the displayed block to include stipple effects between contrasting colours. These effects do not show up properly on a TV, but you can still practise them.

Colour is specified by three parameters in general, but we have so far only used one parameter, and the others have been in default. The second parameter selects a contrasting colour to form a stipple pattern, and the

PARAMETER VALUE	PATTERN NAME	EFFECT
Ø	Single Contrasting Pixel per block of 4.	
1	Horizontal Stripes	
2	Vertical Stripes	
3	Chequerboard	

KEY

PIXEL OF MAIN COLOUR

PIXEL OF CONTRASTING COLOUR

Fig. 8.2. Stipples patterns.

third number chooses the type of stipple pattern to be used. The stipple is formed from groups of four pixels, and the stipple types are shown in Fig. 8.2 for the parameter values. Thus the parameter values 2,4,2 would give red and green combined in a vertical stripe form of stipple pattern.

It would be a good exercise to write a program which steps through all the colour patterns available for a fixed size of rectangle, using 3 nested FOR loops to change the colour parameters. There is a solution which does not use nested FOR loops in the QL manual.

These colour parameters are generally able to be used wherever colour is specified in an argument of a function or statement.

PAPER, INK, BORDERs

As mentioned above, the PAPER is the background against which each window's drawing or printing occurs. The INK is the colour in the printing, and each WINDOW has a BORDER. All these areas are keywords which may take parameters to specify their characteristics.

Try typing in:

```
PAPER 4:INK 7
```

This will change the PAPER colour to solid green, and the ink to white. Nothing happens until a CLS is performed to force the current colouring to alter the output window. Try a CLS now, followed by a PRINT statement with some text to be printed. This will produce a white printout on a green background. Both PAPER and INK take the colour parameters specified above. Even on a TV, you should be able to see the following PAPER stipple. Try typing:

```
PAPER 2,7
```

The default value for the stipple code is 3 which gives a chequerboard. Try adding a third parameter to this statement, and see which stipple effects your TV will show. Now try different parameters on INK. Some PAPER colours will interfere severely with the INK colours; you will soon learn by trial and error which are best.

Channels and WINDOWs

So far we have been restricted to the default windows. These windows can also be changed in size and colour, and more than three windows can also be displayed at a time.

In order to control any entities on a computer, they must be labelled. The labelling of the windows on the screen is by number, and each window is given a separate *channel number* written using the # sign. The channel

numbers of the standard windows are:

command window: # 0
output window: # 1
listing window: # 2

These channel numbers can be specified in the pixel statements given above, as the first parameter. Thus, type in:

```
PAPER #0,2
```

This will change the colour of the command window. Each entry from the keyboard will now be printed as green (the default ink colour) on a red background. INK and BLOCK can take a channel number in the same way, and hence full freedom of use of the screen is possible.

New windows of any size can be defined on the screen, and given their own channel numbers. In addition, statements such as PRINT can also take a channel number, and thus a program may maintain several windows at once, and use each one for a different function, very much as the QL itself does.

To open a new window with a given channel number, the OPEN command must be used. This is a very complex command, and can affect the way in which the fundamental processes of the QL function. We shall meet it again in the next chapter. For now it is used to affect the screen, and to perform this action, it takes a set of parameters to specify the window size and position on the screen. The general form of this command is:

```
OPEN #N, SCR_WIDTH×HEIGHTaHORIZ×VERT
```

The first parameter is the new channel number of the window which you wish to open. The next part of the statement is an identifier of a physical device which the QL recognises – that is, the screen. This is identified by SCR_. Then follows the physical characteristics of the window as shown, with some letters as separators. As an example, try typing in the following:

```
OPEN #9, SCR_50×50a100×100
```

This will open up a new channel of width and height 50 pixels, and with top left corner at (100,100).

Now type in CLS #9 to show up the new window. Note that most of the input and output statements we have met so far should work in this way by specifying a channel number prefixed with #.

Now type in:

```
PRINT #9, "HELLO"
```

The comma after the channel number is generally necessary as a separator. As you can see, the new window acts just like the others. If the word to be printed does not fit on a line, a line-feed within the channel is generated, and the output overflows to the next line. Now try the following, and watch the

window carefully:

```
PRINT #9,"This makes the window scroll"
```

The window acts in every way as for the other windows.

Now try printing over the window using:

```
AT 5,10:PRINT "This overlays #9"
```

Output to any channel will overlay any other channel.

Channel 9 is an output window, and only output commands will work with it. Thus, the INPUT statement is not applicable, and any attempt to use INPUT #9 will give an error. The keyboard is at present attached to the machine at input channel #0. See the section on devices in the QL manual.

While open, the characteristics of a window can be changed with a WINDOW statement. The general form of WINDOW is:

```
WINDOW #N,width,height,horiz,vert
```

The parameters are in the same order as for the OPEN statement, but commas are used as separators.

A window can be closed again by using the statement:

```
CLOSE #N
```

A border can be added to a window of any colour using the BORDER statement. This statement has three groups of parameters. The first is a # number for channel. The second is a thickness parameter in pixels, and the third is the usual group of colour parameters. The border is taken out of the window itself, and thus the total window dimensions remain the same. The general form of the statement is thus:

```
BORDER #N, thickness, colour
```

If no colour is specified, the border is transparent, but still restricts the printing area within that window. For instance, try:

```
BORDER #9,5
```

This puts a 5 pixel thick transparent border within channel 9. Try printing "HELLO" to this channel now to see the effect.

Now try:

```
BORDER #9,5,4
```

This colours the border within #9 green. Try printing again. If you use CLS on the window within which the bordered channel is displayed, the border colour disappears, and can only be retrieved by repeating the BORDER statement. However, the border is still there, and will still restrict the printing.

It is perfectly allowable to define channels within channels. We have defined channel #9 inside #1 in the above. As an exercise, expand window

#9, and define two or three channels within this window, or even overlapping it and the outside. Then add borders, and practise printing to the various channels.

SCROLL and PAN

The contents of a given window can also be made to PAN side to side, or scroll up or down, either in full or in part. Paper is panned or scrolled into the empty space left by the movement.

To experiment with SCROLL and PAN, open a window as below:

```
CLS
OPEN #7, SCR_100x100a50x50
INK #7,0
PAPER #7,5
```

Then use CLS #7 to view the window. The first statement above clears channel 1, which is the normal output channel. The second statement opens a new channel starting at (50,50) and with dimensions 100 by 100. Then the INK and PAPER colours are set within the channel for later printing. If you want a border, use the border command, with a thickness of, say, 5. Do not use CLS on channel 1 again, or the border will disappear.

Now print something in the channel using:

```
PRINT #7, "This is channel 7"
```

This will not all fit on the top line of the channel, and the first three lines will thus be used. To scroll this down the window, we shall use the SCROLL function which has three possible parameters:

```
SCROLL #N, no. of pixels, part
```

The first parameter is the channel number, and defaults to the current output channel. The second parameter is the number of pixels by which the window is to be scrolled, and its sign determines direction. This can be a numeric expression, and if positive will scroll the contents up. A negative value scrolls down. The last parameter refers to the part of the window which is to be scrolled. This may also be a numeric expression, and should only take the values 0, 1 or 2. This defaults to 0, which makes the whole window scroll. If 1 is specified, the part of the window from the top down to but excluding the cursor line is scrolled. If 2, the part from the bottom up to and excluding the cursor line is scrolled. Try the following:

```
SCROLL #7,4,1
```

This will scroll down the top two lines of the window by 4 pixels, and lose half of each character on the second line. The printing on the cursor line is unaffected. Now type in:

```
SCROLL #7,-4,1
```

This will scroll the printing back to where it was, but will not retrieve the contents which fell off the screen during the last scroll.

To scroll the bottom part of the screen, you need to raise the cursor to the top of the window so that there is some printing below it to scroll. The alternative is to use the AT function in #7 to write some data there, and then return the cursor to above that data. Try typing in:

```
AT #7,0,1
```

This will place the cursor at position 0 on row 1 of channel #7. Now the following scroll will act on the part of the screen below the cursor:

```
SCROLL #7,4,2
```

and

```
SCROLL #7,-4,2
```

Try using the scroll command on the command window (#0) to see that you have complete control over the screen. Remember that whether you are in high or low resolution mode, the vertical resolution is the same, and you can, therefore, always scroll by just one pixel, or any odd number of pixels as you wish.

The PAN command is similar, and can act on the whole screen, or part as required. It takes the same number of parameters, with the first two parameters being the channel number, and the number of pixels to move the part being panned left or right. Positive numbers give movement of the contents to the right, and negative to the left. The third parameter defines the part of the screen to be panned, and defaults to 0 which implies the whole screen. Try the following:

```
PAN #7,5
```

and then

```
PAN #7,-5
```

This first moves the contents of the channel to the right by 5 pixels, and the second returns it to its original position.

The third parameter may be added according to the following table:

Value	Effect
0	Pans whole window.
3	Pans whole cursor line.
4	Pans part of cursor line to right of cursor, including the cursor position.

To see the effect, type in:

```
PRINT #7,"123"
```

This moves the cursor to the middle of a line in the window. Then type in:

```
PAN #7,5,3
```

This will move the whole cursor to the right, but if in MODE 8 there is no point in using odd numbers for pixel distances, as the actual number moved is the next lower even number. Thus, while using a TV the 5 in the last statement could be replaced by 4 with the same effect, unless you have specifically changed to MODE 4.

Now type in:

```
PAN #7,-4,3
```

This performs the same, but to the left.

To see part of the cursor line move, type in the following:

```
PAN #7,4,4
```

This moves the cursor position and everything to its right by 4 pixels to the right. Try moving them all back again yourself.

If you are panning stippled colours, you will have to stick to even numbers of pixel movements to retain the effect.

CLS on part of a screen

The CLS function can also be used on part of a specified window by including an extra parameter to define the part to be cleared. The definition of the part parameter is shown in the following table:

Parameter value	Effect
0	Clears whole screen (default).
1	Clears above but excluding cursor line.
2	Clears below but excluding cursor line.
3	Clears whole of cursor line.
4	Clears to right of and including cursor.

To see this, use window #7 from the above, and fill the window with characters using PRINT with a very long literal item. Then use AT to position the cursor in the centre of the window. To see where the cursor is, PRINT a single character there, and bear in mind that it is the next character position to the right which is now the cursor position.

Now type in the following in the order given, watching the screen each time you press RETURN. You will see the relevant portion of the window clear as you go.

```
CLS #7,2
CLS #7,1
CLS #7,4
CLS #7,3
```

Using the part parameter 0 is the same as using CLS without a part parameter.

RECOLour

It is possible to recolour every pixel in a given window using the RECOL command. The 8 parameters of the RECOL statement specify a *solid* colour to which to change the existing colours, in the normal colour order.

Thus the general form of the RECOL statement is:

```
RECOL#N, C1,C2,C3,C4,C5,C6,C7,C8
```

The first parameter defines the colour to change all black pixels to; the second affects all blue pixels; and so on.

Thus, to change all cyan pixels to red in window #7, and leave all the others alone, the following would be typed in:

```
RECOL #7,0,1,2,3,4,2,6,7
```

Try it to see the effect.

FLASH

One other attribute can be given to the display – it can be made to flash. FLASH is only effective in low resolution mode. After FLASH 1 has been executed, all printout to the default channel, #1, from then on will flash. This printing of flashing characters continues until FLASH 0 is encountered. An optional channel number can also be included as for other output statements. For instance, type in:

```
FLASH #9,1
```

Any printout to that channel will now flash.

Flashing characters should not be printed over, as the combination could be unpredictable. Try it out and see the effect. Strange patterns of spurious characters can be the result.

Some further PRINT functions

The size of characters printed can be altered using the CSIZE function. This function takes two parameters, as follows:

CSIZE width,height

The actual effect of this statement varies with the mode. Width may be 0,1,2 or 3, and height may be 0 or 1. The normal sizes are:

CSIZE 0,0 for MODE 4
CSIZE 1,0 for MODE 8

Try running the following program, and then change the mode and run again – you will see that changing the mode does not remove the program, even though the screen clears. This program will show you the available sizes for each mode:

```
10 FOR W = 0 TO 3
20 FOR H = 0 TO 1
30 CSIZE W,H
40 PRINT"THIS IS CSIZE"!W!","!H
50 NEXT H
60 NEXT W
```

Printing can also be backed with a strip of any colour to highlight it. This uses the STRIP statement, which takes parameters of channel number, and the normal colour parameters. The general form of STRIP is as follows:

STRIP #N,colour

The default channel is the output window. Type in the following:

STRIP #0,2

This will back everything printed in the command window with red. Notice, as you type, that only the printed characters are backed with red, the rest of the window is still in the normal paper colour. Try using all the colour parameters as for BLOCK to see the effect of using this statement. A PAPER statement will clear the STRIP colour back to the paper colour, and you will have to re-execute STRIP if you still require a background.

You can further affect the way in which characters are printed by the OVER command. Normally, when a character is printed, its character slot blanks out any character it is printed over. Using OVER you can print characters over others without removing them. The effect is difficult to see on a TV, and you should experiment with OVER 1 and OVER −1 to see the effects. Type in the following program:

```
10 CLS:CSIZE 3,1
20 FOR I = 0,1,-1
30 OVER 0
```

```
40 AT 0,0:PRINT "OVER"!I!"TYPE ANY CHARACTERS"
50 INPUT!C$
60 OVER I
70 AT 5,5:PRINTC$
80 NEXT I
90 GOTO 20
```

When you run the program, the top line will inform you of the current state of the OVER parameter. Start by inputting a few 1s when asked for input. Then the top of the screen will show that you are in OVER 1. Now input some 2s. As you can see, these are overprinted on the 1s without removing them. The next input is for OVER −1, and you can see that the colour of the type changes, and printing becomes very confused. Try running the program again, but just press return instead of typing any letters. This allows the OVER mode to change to 1. Return again will go to −1, and you can type in this mode with a clear paper. As you try the effects, you will see that 0 is normal typing, 1 overtypes, and −1 undertypes in a contrasting colour. Try re-running with changes of INK colour and PAPER colour, and then include some STRIP changes. To see the effects of OVER properly, a video monitor is essential.

The last printing attribute available is UNDER which underlines all printing in the specified channel after the statement has been executed. Try:

```
UNDER 1
```

before running the last program. This switches on the underlining until UNDER 0 is executed again. UNDER can take the usual #N before the 1 or 0 to specify the channel it applies to.

Graphics

We have met two co-ordinate systems so far. The first is the co-ordinate system associated with the AT statement. AT has its *origin* or zero point at top left of the output window, and the points are characters measured from the top and left margins. The second is the pixel co-ordinate system, again with origin at top left, but this time with smaller points, each equal to a pixel.

The next few sections deal with the graphics co-ordinate system. This system can be scaled like graph paper, and you can decide how many points there are vertically, and this automatically adjusts the number horizontally to keep the system square. The origin is at bottom left, in the conventional place for graph paper. The scale for the axes can then be changed at will, and everything drawn after a new scaling will be larger or smaller, and in different places accordingly. The origin for the drawing functions can also be changed to any other point, or to the current cursor position in the window. If co-ordinates are measured from the cursor, this is called 'relative mode'. All drawing on a window is done using the current INK colour for

that window, which may be altered as explained above.

Colours are treated exactly as for pixels, and any window can be specified for output. We shall deal primarily with the normal output window. The default value of the graphics co-ordinate system gives 100 points vertically.

Graphics functions - POINT

If you have set any non-default parameters in the last few sections, you should reset the machine, and start from scratch to experiment with graphics.

Points can be drawn by using the POINT statement with respect to the origin in the bottom left-hand corner of the screen. More than one point can be specified in the statement using the keyword TO. The general form of POINT is:

POINT #N, list of points (X,Y) separated by TO

The point list contains X and Y co-ordinate pairs to label the points. X is measured horizontally, and Y vertically as usual. #N defaults to the normal output window #1. Try the following:

```
10 CLS
20 POINT 2,2
30 POINT 4,4 TO 10,10 TO 40,40
```

The first statement clears the screen to the paper colour, which is red. The second statement plots a point near to the origin in the INK colour. The third statement plots 3 points so that all the points plotted so far are in a straight line at nearly 45 degrees. The accuracy of the 45 degree angle will depend upon the TV or monitor you are using.

The POINT function plots individual points and, as it does so, it moves the *graphics cursor* around. The current position of the graphics cursor is at the last point plotted by any of the graphics functions. This is different from the PRINT cursor, which is affected by PRINT and AT. If you type in PRINT "X", you will see that this is printed at top left where the PRINT cursor starts. Repeating this print statement will move the print cursor along the line in the normal way. The graphics cursor is unaffected, as we shall see shortly.

The POINT function can also be used relative to the graphics cursor instead of to the screen origin. The form of POINT which achieves this is:

POINT_R

This new statement is used exactly as before, but will plot from the cursor wherever it is. For instance, type in:

POINT_R 4,4

This will put a point 4 along and 4 up from the last point plotted, and the graphics cursor will be repositioned to this new point.

CURSOR

The CURSOR statement can be used to move the printing cursor to some point relative to the graphics cursor. We shall see shortly how this allows a graphics pattern to be annotated with print. The CURSOR statement takes the following parameters:

```
CURSOR #N, X,Y(graphics), X,Y(pixel)
```

The first co-ordinate pair is in the graphics system; the second pair is relative to this first pair, and gives the position of the top left pixel of the printing cursor's character slot. For instance, use CLS and NEW and then type in the following:

```
10 POINT 40,40 TO 40,60 TO 60,60 TO 60,40
```

Running this will draw a square of four points on the screen, leaving the cursor at 60,40. Notice that a line number has been used for this statement, though you could simply type it in directly. If you use a line number, and you make an undetected typing error, you can always edit the line without having to retype. This is a distinct advantage for long lines such as this.

If you wish to print the co-ordinates of the points plotted above, printed at suitable places near those points, it would be very difficult to use the AT statement to position the print correctly. We can, however, use the CURSOR statement before the PRINT to cause the printing to start at a suitable place. Try:

```
CURSOR 60,60,2,5
```

This will position the top left corner of the printing cursor at (2,5) pixels from the graphics point (60,60). Now try the following PRINT:

```
PRINT "(60,60)"
```

This will annotate the point perfectly. Now try the same for the other points until you can position the numbers outside the square in just the right place. Try to print the letter O in the dead centre of the square by this method.

If only two co-ordinates are specified for the CURSOR statement, they are taken to be relative to the screen's pixel co-ordinate system with origin in the top left-hand corner. Thus, using CURSOR, printing can be started at any pixel position in any window, and is not restricted to the normal print lines and columns used so far.

LINE

LINE is used exactly as for POINT, but it fills in the line of points between each co-ordinate pair separated by the TO keyword. As usual, it can take a channel number. Try the following:

```
LINE 40,40 TO 60,60
```

The square box drawn using points can now be completed using LINE:

```
10LINE 40,40 TO 40,60 TO 60,60 TO 60,40 TO 40,40
```

Notice that the set of points is closed here; if you leave out the last pair of co-ordinates above, the bottom line of the box will not be present.

The following program draws a set of randomly sized and coloured triangles on the screen starting at (100,75) and fanning out towards the origin.

```
10 CLS
20 REP PATTERN
30 INK RND(8)
40 LINE 100,75 TO RND(50),RND(50) TO
   RND(50),RND(50) TO 100,75
50 END REP PATTERN
```

Some very striking effects can be produced by using the RND function to change INK and position parameters in a simple program such as this.

CIRCLE and ARC

The CIRCLE and ARC functions can be used to draw circular shapes on the screen with respect to the graphics origin. The basic circle function takes the following parameters:

```
CIRCLE #N,centre coords,radius
```

Try typing in:

```
CIRCLE 50,50,10
```

This draws a circle of radius 10 centred at (50,50) in the output window. The circle function can also draw multiple circles by separating several parameters sets by semicolons.

Apart from channel number, two more parameters can be added to turn the circle into an ellipse. The first of these defines the eccentricity of the ellipse, and the second the angle in radians between the minor axis and the horizontal. The eccentricity of the ellipse is the ratio of minor axis length to major axis length. If the eccentricity is 1 (which is the default) a circle is drawn. If the eccentricity is 0, a straight line would normally be drawn at the orientation defined by the angle parameter. However, this does not work on

the early QL machines. Type in the following:

```
CIRCLE 50,50,10,0.5,1
```

This gives an ellipse which fits into the circle above, showing that the radius parameter is the length of the semi-major axis. Early machines will draw some sizes of ellipse at the incorrect orientation. You should experiment for yourself to see the effect of using various different angles and radii before including this function in a program.

The following program draws two sets of concentric ellipses of different colours to illustrate the function.

```
10 CLS
20 INK RND(8)
30 CIRCLE 75,60,56750*RND,0.5,
   2*-1^RND(1 TO 2)
40 GOTO 20
```

The circle function draws ellipses of varying sizes and colours, at either −2 radians or +2 radians. Which is used depends upon a mathematical trick. If the RND in line 30 gives 1, then −1 is raised to the power 1 (see the last chapter) and this equals −1. This is then multiplied by 2, to give −2 radians. If the RND gives 2, then −1 is raised to the power 2, giving +1, which gives +2 radians. This program also shows the speed with which the graphics functions operate.

The CIRCLE statement is also supposed to be capable of use with co-ordinates relative to the graphics cursor using CIRCLE_R. However, this gives an error on early machines, and you should experiment for yourself to find out whether it works.

It is also possible to draw just the arc of a circle, rather than the complete figure using the ARC function. In addition to the usual #N channel number, the ARC statement takes three sets of parameters. You must specify the start point, end point and angle subtended at the centre of the circle of which this arc is a part. The general form is as follows:

```
ARC #N,start point TO end point,angle
```

To gain a feel for using this statement, type in the following:

```
10 CLS
20 INK 0
30 AT 0,0:PRINT"TYPE ANGLE"!!
40 INPUT ANGLE
50 ARC 30,50 TO 60,50,ANGLE
60 GOTO 30
```

When you run this program, you will find that only a certain range of angle values will work. The angle is in radians, and you can only expect it to work between 0 and 2 PI radians, because this is the possible range of values for the angle subtended at the centre of a circle. In addition, on early versions of

the QL, some angles produce strange effects, and the above program will show these up. Try inputting 1 to start with. This will produce an arc below the two points, as the circle is drawn anticlockwise from the start point to the end point. If you want the arc to be above the points, you must swop the end points over in line number 40. If you want to use this function, you should experiment with it using this program.

FILL

Closed convex shapes can be filled with the INK colour using the FILL statement. FILL 1 is used before drawing a shape, and must be switched off by FILL 0 after each drawing has been made.

For instance, RUN the following:

```
10 CLS
20 INK 7
30 FILL  1
40 CIRCLE 30,70,10
50 FILL 0
60 FILL 1
70 INK 4
80 CIRCLE 60,70,10
90 FILL 0
100 CIRCLE 90,70,10
```

Line 30 switches on the FILL function, and the circle in line 40 is filled with white. The FILL is then switched off and back on again to fill the second circle with green. FILL is then switched off again for drawing the last circle. Try removing lines 50 and 60 to see the effect of allowing the FILL to remain switched on throughout.

If you are using FILL for a general shape produced, say, using a multiple LINE statement, then the shape must be convex. Examples of convex shapes are triangles, squares and circles. If it is a re-entrant shape, such as a square with a bite out of it, FILL will not work correctly in general. Re-entrant shapes must be drawn as separate convex shapes and filled separately.

SCALE

As explained above, the default scale of the screen is 100 vertically. This means that a vertical line of length 100 will stretch from top to bottom of the screen. This scale value can be changed to draw graphs of data with any range of values. The SCALE function can also be used to set the origin position. The general form is:

SCALE #N, scale number,(−)origin co-ordinates

These are some examples of the function:

SCALE 10,−50,−50	sets scale to 10, with origin at 50,50
SCALE 0.8,−0.2,−0.2	sets scale to 0.8, with origin at 0.2,0.2

The origin co-ordinates have to be negative, and specify positions with respect to the bottom left-hand corner of the screen. Try this out on your machine to see whether your version requires negative or positive origin co-ordinates.

These SCALE statements can be tested by the following circles respectively:

```
CIRCLE 10,10,10
```

and

```
CIRCLE .1,.1,.05
```

Each of these should give a reasonable circle in their respective SCALEs.

An example

As an example of some of the work above, the following program opens two different windows on the screen, uses one for normal printing, and uses the other in graphics mode to give pairs of random circles, and scroll by one pixel each time. The total effect is one of two processes occurring at once. However, close inspection shows that they are sequential, they are moving alternately. Type in the following and RUN it as a demonstration.

```
10 OPEN #5,SCR_150X150A50X40
20 INK #5,1
30 PAPER #5,4
40 CLS #5
50 BORDER #5,4,6
60 OPEN #9,SCR_190X170A250X30
70 PAPER #9,3
80 CLS #9
90 BORDER #9,4,0
100 N=0
110 N=N+1
120 PRINT #5,"NUMBERS:"
130 PRINT #5,N
140 INK #9,RND(7)
150 CIRCLE #9,25,75,RND(25)
160 CIRCLE #9,50,75,RND(20)
170 SCROLL #9,1
180 GOTO 110
```

You should experiment with this program, and modify it to give different effects. Try overlapping the two windows, for instance.

You can see from this that the graphics co-ordinate system works just as well in this smaller window as it does in the full output screen. Remember that the scaling starts at 100 vertically, so that whatever the physical size of the window, it is divided up into 100 vertical graphics rows. However, in a small window, you will not be able to display 100 separate graphic pixels vertically.

Turtle graphics

This system of graphics uses the screen very much like a plotter. You must imagine a controllable device which carries a pen around the screen and is called a *turtle*. The commands available for the turtle allow it to be moved a certain distance in a given direction, and place its pen on or off the paper to cause it to leave a trail as it goes.

The turtle starts off at the graphics origin pointing towards the right-hand side of the screen, with its pen up. The turtle commands are:

```
PENDOWN
PENUP
MOVE
TURN
TURNTO
```

Each of these can have an optional #N channel number. The first two statements control the pen, which, if down, leaves a trail in the current INK colour as the turtle moves. The third statement needs a distance parameter, and moves the turtle along its current heading, according to the graphics scale in operation. The TURN function causes the heading of the turtle to change by the specified angle *in degrees*. TURNTO allows the heading to be set to an absolute value in degrees. Positive angles are measured anticlockwise from the right-pointing horizontal direction in the conventional manner. To see how this works, you should experiment with angles for yourself.

To start from scratch, reset the computer and try the following:

```
CLS
PENDOWN
MOVE 10
```

The CLS is essential if you are using a TV, unless you set INK to some colour other than white. The MOVE creates a trail 10 graphic units long along the bottom of the screen. Now type in:

```
TURNTO 45
MOVE 10
```

This will show how the turtle can be directed, and then moved. Use TURN

to turn it a further 45 degrees in the positive direction, and make it move again. Can you tell what it will do before you press RETURN?

This graphics system is useful for some games, as well as an output for machine intelligence experiments. A typical application would be to the solution of the problem of finding a way through a maze.

Sound

The QL has a sophisticated sound generation function, BEEP, which takes up to 8 parameters for the full effect. There is another Boolean function called BEEPING which returns TRUE (non-zero value) if the QL is in the process of making a sound, and FALSE (zero) otherwise.

The simplest use of BEEP is to produce a single tone for a specified duration. This is achieved by using two parameters as follows:

```
BEEP duration,tone
```

The duration is an integer variable, and has an upper limit of 32767. This is a number of 72 microsecond periods, and thus the top limit here is just over 2.3 seconds. The value 0 for the duration will switch the sound on until BEEP is executed without a parameter list. The tone is a number from 0 to 255, which goes from high to low respectively. Try the following:

```
BEEP 32000,127
```

This gives a beep for over two seconds, at the mid-range note available. Try changing the numbers here to experiment with the notes obtainable. If you use 0 for duration, type in BEEP to stop it when you wish – you will not be able to stop it with BREAK.

The next parameter that can be added to the list specifies another note. The sound will then bounce back and forth between this and the first note specified. The parameter range for this note is the same as for the first. In addition, two more parameters must be specified to determine the gradients of these changes. The two gradients have stated ranges:

Gradient 1	0 to 15
Gradient 2	−8 to 7

The following will allow you to try out the effects:

```
10 INPUT "1ST PITCH";P1
20 INPUT "2ND PITCH";P2
30 INPUT "1ST GRAD.";G1
40 INPUT "2ND GRAD.";G2
50 BEEP 0,P1,P2,G1,G2
60 PAUSE
70 BEEP
80 GOTO 10
```

When you run this program, you will be able to input the parameters yourself. Line 50 then switches on the beep, and the empty PAUSE statement causes the program to halt at that point until any keyboard key is pressed. The BEEP in line 70 then switches the sound off, and the program repeats. This is a convenient method of testing the range of effects possible with the BEEP function. You could also use FOR statements to step, say, the GRAD parameters through all their values for a given pair of notes. This is a good programming exercise, and you should start the notes at 50 and 150 for the two parameter values respectively.

The final parameters allowed in the BEEP statement are ones which cause the quality of the sound to be controllable. Unfortunately there is little data available on the way in which these work, and it is simply a matter of trial and error, and keeping careful notes of what you do. By this means you will learn to use the function to its fullest extent.

The final parameters are called WRAP, FUZZY and RANDOM. The first two have a stated range of 0 and 15, and RANDOM ranges from −32768 to +32767. You should add these to the program above, and add them in order to the list of parameters in the BEEP statement. As an example of the complexity of the BEEP function, type in the following, and listen to it for at least 15 or 20 seconds. There are clearly several different processes at work within the sound:

```
BEEP 0,50,150,2,-1,8,8,
```

Summary

- The QL screen has a maximum graphics resolution of 512 pixels across by 256 down. In MODE 8, this is reduced to 256 by 256, but with 8 colours. In MODE 4, there are only 4 colours available, with the full 512 by 256 display.
- The pixel co-ordinate system has the origin at top left, while the graphics system has its origin at bottom left. The scale of the graphics system may be set using SCALE. The screen is always adjusted to ensure that the co-ordinate system is square.
- BLOCK is used to draw rectangles in the pixel co-ordinate system.
- Colour may be solid or stippled with two contrasting colours, and can be used in PAPER or INK to set background and printing colours respectively.
- New windows may be opened on the screen, using OPEN and a channel number preceded by #. BORDER is used to give a border to a window. CLOSE is used to close the window again.
- The contents of a window may be wholly or partly cleared, scrolled vertically, or panned horizontally by a given number of pixels.
- CSIZE may be used to change the size of characters.

- OVER causes overprinting, STRIP gives a coloured background to printed characters, and UNDER underlines them. FLASH makes characters flash regularly.
- POINT and LINE are used to draw points and lines in the graphics co-ordinate system. CIRCLE draws circles and ellipses; ARC draws parts of a circle.
- CURSOR moves the printing cursor to a given point in the graphics system for annotating drawings.
- FILL is used to fill convex shapes with the current INK colour.
- Turtle graphics allow a turtle to be moved around the screen, leaving a trail as it goes.
- BEEP is used along with a number of parameters to produce sound from the internal sound generator. BEEPING is a function which returns a true value if the machine is making a sound.

Chapter Nine
Microdrives, Files and Devices

One of the major advances for which the QL is so well known is the inclusion of two disk-like peripheral storage devices – the *microdrives*. These miniature cartridge tape drives will hold about 100K bytes of memory each, and allow a filing system to be used for data and programs. Use of the drives must be attended by some care, and the precautions given in the QL manual should be observed without fail.

The programming of filing and retrieving data is explained, as required for programs which must handle information in any form. Some examples are then given to show how the system works.

The first part of this chapter is all that is needed if you simply want to store and retrieve programs. The rest of the chapter can be read at a later date, when you need to use data files on the QL.

Preparing to use the microdrives

The microdrive cartridges which come with the QL are pushed into plastic outer cases, and should be removed by grasping the fluted end of the cartridge, and pulling out of the outer cover. When you do this, you will see the actual magnetic tape on which all the information is stored. *Do not touch the tape with your fingers under any circumstances.* The cartridge contains a continuous loop of tape, and as the machine winds the tape around, it takes about 7.5 seconds to complete a loop.

There are no distinguishing marks on the tape to tell the computer where the start is, for instance. All this type of information is stored magnetically on the tape, and the first task you have, before using the tapes, is to store this *formatting data* on the tape. There is a special routine called FORMAT which achieves this for you. We will see how to do this shortly.

Figure 9.1 shows the physical layout of the microdrives on the machine, and gives the identifiers used to label them. The left-hand microdrive is drive 1, and will be referred to as MDV1 from now on; the other is MDV2. Each LED (light emitting diode) lights up when its drive is being accessed, thus giving some indication of what is happening.

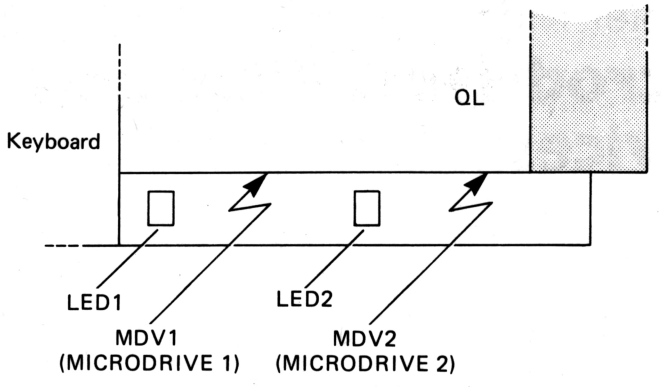


Fig. 9.1. Microdrive layout.

All microdrive functions and statements must include either MDV1 or MDV2. For most activities, they are interchangeable in the functions using the microdrives, but MDV1 has an extra function. When you press F2 or F1 after a reset or switch-on, LED1 will light up showing that the machine is trying to access MDV1. If there is no cartridge present, the machine simply goes straight into SuperBASIC, and allows you to write programs directly. However, if you switch on the machine and insert into MDV1 one of the cartridges supplied with the machine containing a software package, the machine will automatically run that program when you press F1 or F2. Tapes will only fit into the slot in the machine one way round – do not try to force a tape into place. Remember, also, that you must never switch the power on or off with a cartridge in place. This can send an electrical noise-spike to the drive, and destroy the data on the tape.

Try pressing reset, inserting a tape into MDV1 and pressing F1 or F2 as appropriate. The QL loads the program you have placed in MDV1, and LED1 stays on until the process is complete.

The transfer of programs and data from microdrive to computer is called *loading*, and the process of transferring from the computer to the microdrive is called *saving*. You have just auto-loaded a program from MDV1. This means that you did not specifically command the load; the machine took care of the whole process. This chapter is aimed at showing you how to perform the loading and saving of programs and data yourself.

The first task is to format your spare microdrive cartridges – these are the ones which have no writing on the front. The Psion software cartridges have the program name written on them – leave those alone.

Formatting a cartridge readies the tape for use, checks on the amount of space available for storage and imprints a name on the tape for your own identification purposes. FORMAT has the following form:

```
FORMAT MDV1_tape1
```

All microdrive commands use MDV followed by the number 1 or 2, then an

underscore and a name of some kind. The space after FORMAT is essential, and there must be no spaces in the word following the FORMAT. This command will format the tape in MDV1, and imprint it with the name tape1. This name is printed on the screen in subsequent uses of the tape, to allow you to identify it. Try inserting a tape into MDV1, and typing the above. When ENTER is pressed, LED1 lights up, and the tape can be heard to be winding around. This continues until the tape is formatted, which should only take a few seconds. When complete, a message appears on the screen as follows:

```
216/218 sectors
```

This tells you that there are 216 sectors available for use. A sector is 512 bytes, and so 216 sectors give a total of 110592 bytes of storage. The usual notation for memory capacity is in K bytes, and 1K equals 1024 bytes. Thus, 216 sectors is 108 K of storage.

Place another tape in MDV2, and type:

```
FORMAT MDV2_tape2
```

This will use the other drive. You should now repeat the format for each tape about six times to bed the cartridges in. You should do this with all new tapes.

The other microdrive commands have a similar form to FORMAT, as we shall see.

DIRectory command

The microdrives store data onto the tape in blocks which may contain programs or data. Each block is called a file, and is labelled with an identifier. These identifiers are called file-names. The complete list of file-names of files stored on a tape is kept in a special area of the tape called the *directory*. To view the directory of a tape, you can use the DIR command, which has the form:

```
DIR MDVN_
```

where N is 1 or 2 on the basic machine. More microdrives can be added to the machine, and these will be given numbers from 3 onwards. Try inserting one of the Psion tapes in MDV1, and type the following:

```
DIR MDV1_
```

This will cause the tape to start up, and after a short delay a message comes up on the screen with the name of the tape, the number of sectors free, and a list of the files on the tape.

If you stored the game program given in an earlier chapter, you will be able to use DIR to check on the file stored on that tape.

Devices

The microdrive is an example of the various Input/Output devices which are controlled by the QL. The word *device* in SuperBASIC means any one of the following:

SCR - the screen.
CON - the console, which includes the screen and keyboard.
MDVN - microdrive number N.
SERN - serial port number N.
NET - the network.

We have seen how OPEN may be used on the SCR device to define a special window on the screen. Recall that OPEN defined a #N (or channel) number which was used by PRINT, and other statements, to communicate with the window. The same philosophy is used to contact any of the other devices. For instance, you can store information on one of the microdrive devices by using an OPEN statement to give it a channel number, and then sending data to it with PRINT exactly as if it were a window. We shall see this shortly.

Some of the devices above have default settings as to their characteristics, and you should consult the QL manual if you are interested in experimenting with these.

The SERN device, where N is 1 or 2, is a channel which communicates in a special format called *serial*. A description is beyond the scope of this book, but with the correct cable, and use of the BAUD statement, it can be used to operate a printer. Once again, the normal PRINT statement with a # number opened to be a SER device will perform the printing.

Saving, loading and running programs

To experiment with the microdrives, you will need a program to save and load. Type in the following program:

```
10 REM TEST_PROG
20 CLS
30 PRINT"TEST_PROG"
40 PRINT"LINE 40"
50 PRINT"LINE 50"
60 PRINT"LINE 60"
70 PRINT"LINE 70"
80 PRINT"LINE 80"
90 PRINT"LINE 90"
```

We will now use the SAVE function to save the program to a tape, and the LOAD statement to recall it. You will need to insert a formatted tape into MDV1 for the following.

The SAVE and LOAD statements apply to SuperBASIC programs only, and not to general files of data. The statements need some parameters to work correctly. The most important parameter is the actual device onto which the saving or loading is to occur. The general form of the SAVE statement is as follows:

```
SAVE device_filename,line list
```

There are actually two parameter areas here. The first is the device and filename, and the second is a list of line numbers to be saved. The first of these would be of the form:

```
MDV2_test_prog
```

No spaces are allowed in this word, and you will have to use underscores to separate words in the filename if you need this type of identifier. For instance, TEST PROG is not a valid filename, but TEST_PROG is. This is shown above. Note that as with all identifiers in SuperBASIC, lower- and upper-case letters are treated as identical. MDV2 is the device name, and must be connected to the filename by an underscore character.

The line list is the other parameter area required with SAVE, and it is shown separated from the device and filename by a comma. This comma may not always be necessary, but it will save bad line errors. The form of the line list is identical to that of the LIST function. If omitted, the whole program will be saved, while a list of lines separated by commas, and ranges using the TO separator, will save parts of the program. For instance, the following (do not type this in):

```
SAVE MDV2_test_prog, TO 40
```

will save all the lines of the program up to and including line 40. See the LIST statement, in a previous chapter, for all the other alternatives.

To see this program saved on the microdrive, type in the following:

```
SAVE MDV1_TEST_PROG
```

This will file the whole program under the name TEST_PROG onto the tape. To see that it has arrived there, type in:

```
DIR MDV1_
```

Now type in the following:

```
SAVE MDV1_TEST_20_60, 20 TO 60
```

and then:

```
SAVE MDV1_TEST_70_90, 70 TO 90
```

This will put two further files on tape, called TEST_20_60 and TEST_70_90, with part of the program in each as suggested by the

filenames chosen. Do a DIR to see the full list on tape, and to see how much storage is left.

We can now see how to LOAD a program from tape. The LOAD statement only takes the parameter of device_filename. When a LOAD is executed, it first performs a NEW, which wipes out any existing programs, and then loads in the contents of the stated file. Try listing the program in memory at the moment before continuing. Now type in:

```
LOAD MDV1_TEST_20_60
```

and then LIST the program again. You will see that the program has been replaced by the part program of lines from 20 to 60. Now do the same for the other part program, and then for the whole program to gain a feel for the function.

The QL error checking is still in action during a load, and each line is checked for correct syntax as it is loaded. If there is an error, the word MISTAKE is inserted at the start of the line, and an error comes up when the program is RUN. This is not so important for programs which have been saved after typing in from the keyboard, as these are checked then. However, it is very useful if loading is occurring from another device, or from a program file created and stored by a program.

We can now save and load program files to any device simply by specifying the device name. An example of this would be:

```
SAVE SCR
```

This takes the current program in memory, and sends it to the screen device. It will display it there accordingly – try it!

If you have a printer connected to the serial port, using SER_ as the device name will send the program to the printer. You will need to use the BAUD statements correctly to achieve this – see the printer and QL manuals.

DELETE

When you wish to remove a file from the tape, you can use the DELETE statement which requires a device name and file name as for the above statements. Try DIR to remind you of the files on MDV1, and then type in the following:

```
DELETE MDV1_TEST_20_60
```

This will remove the file stated, and you should do a DIR to see that it has been successful. Be very careful with this statement, or you will lose important data.

You may have noticed that sometimes, while using the above statements, the cursor reappears before the tape has stopped winding forward. You can

continue typing at this point, even though the tape is still going, as the keyboard is active whenever the cursor shows it to be ready.

LRUN, MERGE and MRUN

A program can be RUN directly after loading using the LRUN statement. This is short for LOAD and RUN. Simply specify the device and filename. This statement will only work for a complete program, unlike LOAD which allows part of the program to be brought in.

For instance, try the following:

```
LRUN MDV1_TEST_PROG
```

If you had another program in memory before this was typed in, it will have been lost as LRUN starts with a NEW command. Again, LRUN checks the incoming lines for syntax errors, and writes MISTAKE if it finds any.

If you do not wish to lose the program in memory when loading from a device, you can use the MERGE and MRUN statements. These bring in lines from the file, overwriting any of the same line number, but leaving others alone. To see this at work, type in and save the following program under the filename OVERLAY:

```
60 PRINT"REPLACES OLD LINE 60"
100 PRINT"THESE ARE EXTRA LINES"
110 PRINT"TO BE ADDED TO"
120 PRINT"TEST PROG."
```

Use DIR to see the files on the tape.

Now type in NEW to clear the memory of programs, and confirm this with LIST. Bring TEST_PROG back into memory using LRUN, to ensure that it has arrived. Now type in the following:

```
MERGE MDV1_OVERLAY
```

This will merge the lines of OVERLAY with TEST_PROG, overwriting where necessary. If you now RUN the program, you will see this effect. Use LIST to see the merged program.

To RUN the merged version immediately after loading, use MRUN, which is short for MERGE and RUN. You can try this by typing in the following:

```
MRUN MDV1_TEST_PROG
```

This will bring in TEST_PROG, and RUN the merged result. The only difference will be at line 60, as you can see.

Both MERGE and MRUN check for syntax errors and treat them as for LOAD.

COPY and COPY N

COPY is used to transfer files from one device to another. This could be used, for instance, to copy a microdrive file from one microdrive to another, or from a microdrive to another device.

When information is stored on a microdrive, a special packet of data called a *header* is stored first, followed by the information to be stored. The total file on tape is then identified for later recall by the header information. If you wish to send a file of data to a non-microdrive device, it is useful to be able to send it without the header, which would otherwise interfere. This is the function of the COPY_N statement.

We shall perform two copies, using COPY, as examples of the process, and the first is a file copy from one microdrive to another. For this experiment, you will need a further formatted cartridge in MDV2.

The general form of COPY and COPY_N is as follows:

```
COPY device_filename TO device(_filename)
```

The second device above will need a filename attached if it is a microdrive. The first file will then be transferred to the second drive, under the new filename, which can be the same as the first if you wish.

Try the following:

```
COPY MDV1_TEST_PROG TO MDV2_TEST2
```

A DIR command will show that the tape in MDV_2 has a new file called TEST2 containing the program in TEST_PROG. This can also be used to make another copy of TEST_PROG on the same drive, if you wish. See if you can do this, calling the second copy TEST2. Then LRUN it to check it works.

COPY is useful for backing up important files. You should always take an extra copy of files you wish to keep. These backup copies should not be used unless the main copy should fail. Then the backup should be used purely to produce another main copy.

Another example of COPY would be to copy a file directly to the screen. This can be done as follows:

```
COPY MDV1_TEST_PROG TO SCR
```

Try it to see the effect. You do not need to use COPY_N for this copy, but using COPY_N will have the same effect.

Some further commands

There are several commands which are beyond the scope of this book, and are orientated towards machine code programming. They are LBYTES, SBYTES and SEXEC. To use these commands you will have to understand

how the memory of the QL is arranged, and the type of data which is stored there. These commands are used for storing and saving binary data directly.

Another very interesting command is the EXEC command which allows a type of program execution called multi-tasking. Under this regime, several machine code programs may be set running simultaneously within the machine, each with its own input and output area, perhaps as separate windows.

Another important command is the NET command. This allows a network of QLs and Spectrums to be connected together for fast communications, and sharing memory.

Files and file handling

There are several types of file which can be stored on a microdrive. We have seen program files already. This section just deals with the sequential data files which can be stored and retrieved by a SuperBASIC program.

One of the most useful attributes of a computer is its ability to manipulate, store and retrieve large quantities of data. For instance, you may want to store several hundred stock items in a business, each with its own label and price. When a stock item is requested, you may want to be able to type in its number, retrieve its price, and have a full description printed out on a receipt. This entails the storage of data on peripheral memory such as a microdrive cartridge. When the stock cartridge is removed, and another inserted, you may want to be able to access a file of customers' names and addresses. Another file needs to be set up, and the data retrieved in a particular way. Another application would be in scientific experiments where a file of outcomes of a set of trials may need to be kept. Again, peripheral storage is essential. This section shows you how to store and retrieve raw data within your programs.

You should think of a file as a set of groups of data items, separated by line-feeds. Each group is called a *record*. Each record can be thought of as a single line of numeric and string variables. For instance, in a telephone directory, a record would consist of a name, an address and a telephone number. The next subscriber would be printed on a new line, and occupy a new record. There are advantages to simply storing each character in a file on a separate line, and hence in a separate record. You will see that the minimum amount of data which can be read from a file is one record. This means that if you want to retrieve separate characters from the file, each must occupy a separate record, or string slicing must be used.

To define a file, the OPEN_NEW and OPEN_IN statements are used. The general form of these statements is:

```
OPEN_NEW #N,device_filename
OPEN_IN  #N,device_filename
```

The first of these opens a new file called filename on the stated device, and assigns a channel number N to it. This file is available for writing to. The second statement opens an existing file for reading from. The channel number is used by PRINT and INPUT just as if these files were windows. When opened for writing, a new file is empty, and PRINT is used to fill it with data. The process of using a file is only complete when a CLOSE statement is encountered for that channel number. There are many internal manipulations which are needed for filing, and some of them are only completed at the end of the process, CLOSE finishes them off.

The following program puts one record into a new file called DATA1 on MDV1:

```
10 OPEN_NEW #7,MDV1_DATA1
20 PRINT #7, "THIS IS ONE RECORD"
30 CLOSE #7
```

The PRINT statement used for filing records on a file will store that data exactly as it is normally printed. Each line of print is stored on each record in the file.

After this program is RUN, the DIR statement will confirm the existence of the file, and the COPY statement confirms that it contains the required data. Try the following:

```
COPY MDV1_DATA1 TO SCR
```

You can always check on a file by this method.

As you can see, the same commands that are used for program file storage also work for data file storage. Recall that if you try to LOAD a file containing bad lines, a mistake or error will be shown up. Try LOAD MDV_DATA1 to see the effect.

Notice that the # number chosen for a file is not used in the file itself. The channel number is purely internal to the program that creates or reads the file, and can be a different number in the creation program from that used during reading. Any number, up to 15, could have been used as long as it does not equal one of the default channels such as the output window, or the command window.

To read the data in DATA1 back into a program, the OPEN_IN statement is used to define the file as open for input. Try the following:

```
10 OPEN_IN #5, MDV1_DATA1
20 INPUT #5, A$
30 CLOSE #5
```

This will open the file DATA1 for reading, input the first (and only) record stored there into A$, and then close the file. To see that it has worked, type in PRINT A$.

If you try to open a new file for writing which has the same name as one already on file, an error message comes up on the screen. This shows you that you must always create a new file for writing out data, even if you would

prefer to update an existing file. To add to an old file, its data must be read into the new file, and then the new data added. This is cumbersome, but essential in sequential access filing.

When more than one item is stored on a line, as when more than one item is printed, the normal separators are used, and the data will appear in the record according to the normal printing format of the line. For instance, the following program stores 10 pairs of numbers in two columns, eight spaces apart, using the normal comma as a separator:

```
10 OPEN_NEW #6, MDV1_NUMPAIRS
20 FOR I=1 TO 10
30 PRINT #6, I,I+1
40 NEXT I
50 CLOSE #6
```

After RUNning this type in COPY MDV1_NUMPAIRS TO SCR to see the data stored. This will confirm that the data is stored in each record exactly as printed by the PRINT statement. The following program lines, added to the above, will input the data into the variable D$:

```
60 STOP
70 REM
80 REM Now retrieve the data:
90 REM
100 OPEN_IN #7,MDV1_NUMPAIRS
110 FOR I=1 TO 10
120 INPUT #7,D$
130 PRINT D$
140 NEXT I
150 CLOSE #7
```

The STOP in the first line is purely to separate the two programs so that the first will not run straight into the second when executed.

When the second program is run using RUN 100, the record lines are recalled from the file, and read into D$. They are then printed one by one.

To store the numbers separated by a single space, change line 30 to:

```
30 PRINT #6,I!I+1
```

The two programs can only be RUN after the present file NUMPAIRS has been deleted using DELETE.

As you can see, the numbers above are treated as strings. To treat them as numbers, a numeric variable is used in line number 120 when they are read back. With the data records in NUMPAIRS, only the first number of each pair is read into a numeric variable, as a space is taken to be the end of each number, and anything else on the line is disregarded for INPUT. You may have noticed this in the normal use of the INPUT statement from the keyboard.

To store numbers in a file, each number must be stored in a separate

record, which means that the PRINT statement must PRINT each successive number to a new line. Try changing line 30 to:

```
30 PRINT #6,I\I+1
```

This will store each number to a new line, i.e. to a new record. Now change lines 120 and 130 to:

```
120 INPUT #7,D
130 PRINT D
```

Remember that NUMPAIRS must be deleted before the two programs are run, or the first program will cause an error, as it will be trying to store a file the name of which already exists.

When you RUN the first program, at line 10, a file is produced with each number output to a new line, and hence a new record. There are 20 numbers. You can see this by using COPY. The second program, at 100, just takes the first 10 of these numbers, and loads them into the numeric variable D, and prints them.

To read in more than one record at a time, more variables can be placed in the INPUT statement exactly as for keyboard entry. Thus you could change lines 120 and 130 to:

```
120 INPUT #7,D,E
130 PRINT D!!E
```

This will INPUT and PRINT all 10 pairs of numbers stored in NUMPAIRS.

As you can see, you can create files of any type and configuration by using PRINT correctly.

A couple of final words of warning are due at this point. The early manuals are at variance with some of the above, and you are advised to try out all the examples to convince yourself of the methods stated here. The other point concerns the early version of the software in the QL. Sometimes, when an MDV reference such as DIR is made, the microdrive does not start up, but the action appears correct. This is a case of the software working from an internal copy of the MDV data without referencing the drive. This does not cause any errors if you do not touch the cartridge in use, but if you place a new cartridge in the drive, the data stored in memory will be out of date. Make sure that the drive makes a noise each time a microdrive reference is made, of any type. If it does not, try a DIR to the other drive, even if empty; this can jog the system back into action. If this does not work, and you do not want to reset, try using COPY in a non-destructive attempt at returning to normal.

Summary

- FORMAT is used to prepare a microdrive tape for use.
- The microdrives are called MDV1 and MDV2 (left and right respectively).
- DIR shows the tape's directory of files.
- There are five groups of *device* in QL SuperBASIC, the MDVs being one. SCR is the screen, CON is the console which includes screen and keyboard. SER1 and SER2 are the serial ports. NET is the network channel.
- OPEN is used to apply a channel number to a device so that statements such as PRINT and INPUT can interact with data on those devices.
- SAVE, LOAD, LRUN, MERGE, MRUN are used for the storage and retrieval of program files on microdrive. When a program file is recalled using these statements, its lines are checked for correct syntax, and an error results if a line is incorrect.
- DELETE is used to remove a file from a microdrive tape.
- COPY is used to copy data from one device onto another. It can be used, for instance, to produce backup copies of important files.
- Make at least one extra (backup) copy of all important files in case a malfunction erases the tape.
- Data files are held sequentially on tape, and must be opened for reading in using OPEN_IN or new files opened for writing out using OPEN_NEW. An existing file cannot be altered once closed. CLOSE must be used at the end of a file's use to ensure that all the actions on that file are properly completed.
- Data is written and read to and from files using PRINT and INPUT as if they were screen and keyboard respectively. Each printed line is called a record.

Chapter Ten
Procedures, Subroutines and Functions

This chapter describes the ways in which the list of SuperBASIC keywords can, effectively, be extended to include your own commands and functions. It also shows how you can write part of a program as a module which can be placed at the end of the main program, and called up as many times as required throughout the main program. This saves having to write a section of program several times if it is to be used more than once.

The ideas of this chapter allow programs to be modularised so that as they become more and more complex, their structure can remain comparatively simple. The concepts involved can be difficult to assimilate, but by learning to use these facilities, you can produce easily modified and neatly structured programs which allow even the most complicated applications to be broken into small, easily managed parts.

Procedures

A *procedure* is a separate block of program, which can be placed at the end of the main program and is called up in a similar way to any of the standard SuperBASIC statements or functions. Variable values may be passed to and from the procedure as required. Procedures are useful for parts of a program which have to be used more than once.

Consider the problem of checking whether the answer to a question is yes or no. We have encountered this problem in a number of forms so far, and each time this function has been carried out in the main part of the program. To some extent this function is a side issue to the main function of a program, and it would be neater if it could be placed out of the way and called up as necessary. The main program would then appear less cluttered, and be easier to read and write. We could write this function as a procedure, stored as a separate module. This philosophy helps to structure programs in a straightforward and readable manner.

A procedure block is defined using the DEF PROC statement. It is followed by the name of the procedure, and a set of parameters *declared*

within brackets, which are to be passed to and from that procedure. A simple example would be:

```
DEF PROC ADD(X,Y)
```

This is a procedure called ADD, which takes in the parameters declared as X and Y. The actual body of the procedure is written next, ending, typically in SuperBASIC, with END DEF. This procedure might simply add X and Y, and the full procedure could look like this:

```
100 DEF PROC ADD(X,Y)
110 PRINT X+Y
120 END DEF
```

This is a separate block of program statements which is only executed when the procedure is called up. Whenever this procedure is called, it must be given the values of X and Y, and it will then proceed to print their sum. The procedure is called up using its name directly, followed by the parameters in the same order as in the brackets of the DEF PROC statement. To call this procedure for the numbers 5 and 6, the following statement is used:

```
ADD 5,6
```

When you call a procedure for execution, you must ensure that the values following the name are in the same order as those in the bracketed declaration of the procedure definition.

As you can see, once a procedure has been defined, calling it is simply a matter of writing its name, followed by any parameters it may need. On many occasions there will not even be any parameters. For instance, take the example of a standard screen message which you may need printed several times. The procedure to perform this action might be:

```
100 DEF PROC MESSAGE
110 PRINT FILL$("*",26)
120 READ MESS$
130 PRINT MESS$
140 PRINT FILL$("*",26)
150 DATA "This is the first message"
160 DATA "This is the second message"
170 DATA "This is the third message"
180 END DEF
```

As a routine, this is fairly simple, but it is complex enough to clutter the main program to a considerable degree, especially if it is used several times. Here it simply resides at the end of the program, out of the way, and is called for execution by using just the name MESSAGE. A program using this procedure three times could be of the following form:

```
10 CLS
20 RESTORE
30 MESSAGE
40 INPUT A$,B$,C$
```

```
50 MESSAGE
60 PRINT A$,B$
70 MESSAGE
80 PRINT C$
```

This program is extremely simple to read, and would be followed by the 9-line procedure definition above. Each time the command MESSAGE is executed in the above program, the computer recognises that this is not a normal SuperBASIC keyword, and it searches through the program to find a procedure with that name. It then executes the procedure, and returns to the line following MESSAGE.

Before seeing a full example of the above, we shall look at how to pass information back to the main program from a procedure.

Suppose the two numbers X and Y are to be added, but the sum is not to be printed directly. It is to be returned to the main part of the program, to have a third number added to it from the keyboard. The sum will be stored in the variable SUM, which must be included in the brackets following the procedure definition, as well as in the call to that procedure in the main program. Thus, we might write:

```
10 REM - main program:
20 CLS
30 ADD 15,38,SUM
40 INPUT"next number"!Z
50 PRINT"answer ="!SUM+Z
60 REM
70 REM - now for the procedure:
80 REM
90 DEF PROC ADD(X,Y,SUM)
100 SUM = X+Y
110 END DEF
```

In line 30, the procedure call includes a couple of values to be added, and a variable, SUM, to which the result is to be returned. When this is executed, the procedure with the name ADD is executed, which includes an assignment for the variable SUM. Thus line 30 causes SUM to be evaluated so that its value is available for use in subsequent lines. In particular, line 50 uses it to add to the number input from the keyboard. The call to the subroutine can include numbers and/or numeric expressions as long as they are written in exactly the same order in both the procedure call and the procedure definition.

So far we have carefully used different variable identifiers in the main program from those used by the procedure, except when they have to share the same identifiers for passing values to and from the procedure. In fact, variables declared in a procedure definition are called LOCAL varibles; they do not interfere with the same variable names used in the main program unless they are mentioned in the procedure call of the main program. For instance, X and Y are mentioned in the declaration of the procedure in line

90, and from then on they are considered to be LOCAL. SUM is also LOCAL, but it is specifically brought out of the procedure by being included in the procedure call in line 30. As X and Y are fully LOCAL, you could use them in the main program for other purposes, and they would be unaffected by execution of the procedure. The computer carefully stores them away when the procedure is entered, and restores them again when the main program is re-entered. Try PRINT X and PRINT Y after running the above program, you will see that they are considered to be unused variables. However, if you PRINT SUM, its value is still stored.

Variables used within the procedures, but not actually declared in the DEF PROC statement are not local, and their values are still held when the main program is re-entered. Such variables are called *global* variables. In addition, all variables used in the main program are global, unless they specifically become LOCAL variables to that procedure.

The following gives an example of a variable (K) not declared in the DEF PROC which is thus global:

```
DEF PROC NUMBER(A,B,C)
K = A-B*15
C=18^K
END DEF
```

This procedure has three LOCAL variables, A, B and C, because they are declared in the DEF PROC. However, K is global. If the main program is in the process of using a variable with this name, its value will be interfered with by this procedure. Of course, the variable C above is probably needed in the main program, and it will be brought out by being included by name in the following type of procedure call:

```
NUMBER 23,D,C
```

Here, the values 23 and D are sent to the procedure, while the value of C returns.

In order to ensure that there is no unauthorised interference between variables in a procedure, and variables in the main program, the LOC statement can be used to make a variable LOCAL. For instance, the following could be included at the start of the above procedure:

```
LOC K
```

K would then be as LOCAL as A and B.

To write well structured programs, it is worth following a few simple rules, though it may incur a little extra effort. For instance, it is not a bad habit to use LOCAL as a matter of course for any variables used in a procedure which are not actually declared. In a simple program you can avoid any problems without this. In a more complex program, which you might wish to adjust at a later date, using LOCAL can save you from having to analyse each procedure to check that you are not interfering with any new

variable names you want to use in the main program. If you always use LOCAL in this general way, you will know that all your procedures are safe. In the same way, you should always place your procedures at the end of the program, with a REM statement to say that the procedures start there. Again, this leads to well-ordered, easily modified programs. Treat procedure writing as a method of transferring often used routines to an area where, once created, they do not need to be worried about again. Using procedures makes the main program much simpler, and it is not uncommon for a medium-sized program to have fewer program statements in the main part of the program than in the procedures.

It is also a good idea to make procedure names as meaningful as possible. A procedure is used in a similar manner to a BASIC command, and one of the great advantages of BASIC is its nearness to English. Make your commands look like English statements to gain the same type of advantages.

An example

The example in this section uses a number of ideas from the previous work, and has a number of procedures. It simply requests the three coordinates of a triangle, draws it, and then asks if you want it filled in, and with what colour. The main program is contained in a loop called PROGRAM, so that at the appropriate point an EXIT statement can be used to exit the loop and finish the program. The program also uses the real time clock in the QL, and you will have to set it before running the program.

```
10 REM - Main Program
20 REP PROGRAM
30 CLS:INK 7:STRIP 2
40 HEADER
50 PRINT
60 PRINT"PLEASE TYPE THREE SETS"
70 PRINT"OF COORDINATES."
80 PRINT
90 STRIP 5:INK 0
100 INPUT"X1  Y1"!!X1!!Y1
110 INPUT"X2  Y2"!!X2!!Y2
120 INPUT"X3  Y3"!!X3!!Y3
130 DRAW_TRIANGLE X1,Y1,X2,Y2,X3,Y3
140 AT 0,4:CLS 1
150 STRIP 7
160 HEADER
170 PRINT:PRINT"FILL IN?"!!
180 YES_OR_NO ANSWER
190 IF ANSWER THEN
200 INPUT "COLOUR? 0 TO 7"!!C
210 INK C
```

```
220 FILL 1
230 DRAW_TRIANGLE X1,Y1,X2,Y2,X3,Y3
240 FILL 0
250 INK 0
260 END IF
270 AT 0,4:CLS 1
280 STRIP 5
290 HEADER
300 PRINT:PRINT"RUN PROG AGAIN?"!!
310 YES_OR_NO ANSWER
320 IF NOT ANSWER THEN
330 PRINT"BYE BYE"
340 EXIT PROGRAM
350 HEADER
360 END IF
370 END REP PROGRAM
380 REM
390 REM - PROC DEFS start here:
400 REM
410 DEF PROC HEADER
420 AT 0,0:
430 PRINT"*****"!DATE$!"*****"
440 END DEF
450 DEF PROC
    DRAW_TRIANGLE(X1,Y1,X2,Y2,X3,Y3)
460 LINE X1,Y1 TO X2,Y2 TO X3,Y3
    TO X1,Y1
470 END DEF
480 DEF PROC YES_OR_NO(ANSWER)
490 LOC ANS$
500 INPUT ANS$
510 IF ANS$(1)=="Y" THEN
    ANSWER=1 ELSE ANSWER=0
520 END DEF
```

To follow the working of this program, you should first scan through and find the main loop. This starts at line 20, and ends at line 370. The EXIT for the loop is at line 340 under certain circumstances. Within the loop is a number of procedure calls. They are all called by names which suggest their function, and each has a procedure definition in the last section of the program which starts at line 390. There are three procedures, as follows:

HEADER	Prints a header on the screen, including date.
DRAW_TRIANGLE	Draws the triangle with co-ordinates X1,Y1; X2,Y2; X3,Y3.
YES_OR_NO	Inputs a question and checks for YES, or anything else starting with upper- or lower-case Y, using the == relation. If true, the variable

ANSWER is set TRUE, by giving it a non-zero value. If FALSE, ANSWER is set to FALSE, by value 0.

Line 30 of the program clears the screen, and then sets INK and STRIP to the normal default values. This is necessary as the program adjusts these values, and does not reset them. This means that later runs of the loop would start off differently from the first pass. Line 40 calls the procedure HEADER.

The next part of the program asks for the input of three co-ordinates, which are then used in the DRAW_TRIANGLE procedure. Line 140 is a method of clearing the top few lines of the display before the header is printed again, with the latest time, followed by the question FILL IN? which requires a YES or NO answer. This question is handled by the YES_OR_NO procedure. If the answer is in the affirmative, the colour must be put in, and DRAW_TRIANGLE is called again to fill in the triangle. You are then asked whether the program is to be run again, and YES_OR_NO called again to handle the interaction. The program loop then runs again, or stops there.

As you can see , the procedures are all used more than once, and trying to achieve this using GOTO or REP etc. instead of procedures would be most cumbersome.

You should type this program in, and then save it on microdrive. You can then spend some time modifying it to change the screen printout, and adding more tasks in the form of procedures.

Subroutines

To retain compatibility with other forms of BASIC, the QL allows the statements GOSUB and ON GOSUB with RETurn to be used to produce 'subroutines'. A *subroutine* is simply a block of program, starting at some line number, and ending with a RETurn statement. The block is executed by GOSUB followed by a line number. For instance, the following shows a program which prints three different messages, using the same subroutine each time:

```
10 CLS
20 A$="First Message"
30 GOSUB 200
40 A$="Second Message"
50 GOSUB 200
60 A$="Third Message"
70 GOSUB 200
80 STOP
200 REM - SUBROUTINE STARTS HERE
210 PRINT FILL$("*",14)
```

```
220 PRINT A$
230 PRINT FILL$("*",14)
240 RET
```

Again, the subroutine contains a block of code which is called whenever required. When a GOSUB to the appropriate line number is made, the computer stores its present position in the program, and goes off to the subroutine. When the RET is encountered, the program is continued where it left off. Subroutines can be used to provide the same degree of modularity to a program as procedures, but are less sophisticated, as you can see.

The ON GOSUB statement works exactly as for ON GOTO, and both are replaced in SuperBASIC by the SELection statements.

RET can also be used in procedures to force a return to the main program, perhaps in an IF statement if its condition is satisfied.

User-defined functions

Just as it is possible to add your own commands using procedures, you may also add your own functions in SuperBASIC.

Functions are similar to procedures in that they contain a common piece of programming which is required often, but a function differs in that its identifier is regarded as a numeric expression in calculations. For instance, in a previous example in this chapter, we defined a procedure called ADD. We could never write a statement of the form:

```
N =3*ADD(X,Y)
```

because ADD is not a function. On the other hand, we could write:

```
N = 3*SQRT(X)
```

because SQRT is defined as a function. We can add our own functions to the list of SuperBASIC functions by using the DEF FN statement. Just as for procedures, this has to be followed by a name and brackets within which are declared the variables upon which it acts. The DEF FN statement is then followed by a set of statement lines which define the evaluation of the function, and these end with END DEF. The RET statement is used in the function evaluation to return the required value. This is illustrated below.

The following defines a function called ADD which returns the sum of four numbers when used as a numeric expression:

```
100 DEF FN ADD(X,Y,Z,T)
110 A=X+Y
120 B=Z+T
130 RETurn A+B
140 END DEF
```

Note the similarity between this and a procedure. However, a function returns just one number, and it is stored under the name of the function. The single value returned by the function is calculated and identified in line 130 by the RET statement.

Function names must have the correct type symbols. Here ADD is a floating point valued function. If string valued, it would have to be ADD$, and if an integer ADD%.

Just as for procedures, a variable used within the definition of a function can be kept local by using the LOC statement, and it would be worth adding the line:

```
105 LOC A,B
```

to keep the function from interfering with any possible use of the same variables in a main program.

The ADD function could be used in a numeric expression in the main part of a program by adding the following lines to the above:

```
10 CLS
20 INPUT "4 numbers:"!X!!Y!!Z!!T
30 K=123*ADD(X,Y,Z,T)
40 PRINT K
```

This gives values to four variables, and then sets K equal to 123 times the value of the function ADD with argument as shown. As you can see, the ADD function is used exactly as for the standard functions which come with SuperBASIC, and you can thus add your own functions to the main set as you wish.

Summary

- A *procedure* is a block of statements called into execution by a single command in the main program. The procedure starts with DEF PROC and ends with END DEF. Variables within the procedure are local if declared in the procedure heading, or if mentioned in a LOC statement. Variable values may be passed to and from a procedure.
- A *subroutine* is a block of program, terminating in a RET statement, which is called by the GOSUB statement followed by a line number.
- DEF FN and END DEF are used to define new functions which are used in exactly the same way as the normal repertoire of SuperBASIC functions. A function differs from a procedure in that a function returns exactly one number which is stored under the function's own identifier. Function identifiers are assumed to be floating point unless ending in $ or % for string and integer respectively.

Appendix

Alphabetic List of Keywords, Functions and Operators

This appendix supplies a complete list of the QL SuperBASIC keywords, in alphabetic order. Each keyword is indexed by chapter, and has a short description to remind the reader of its function. The list of other operators and functions which include, for instance, + and * will be found in the concepts section of the QL manual itself.

Keyword	*Chapter*	*Type*	*Description*
ABS	7	Function	Absolute value
ACOT	7	Function	Arc cotangent – gives radians
ADATE	4	Statement	Adjust clock by seconds
AND	5	Operator	Boolean AND
ARC	8	Statement	Draws arc of circle
AT	4	Statement	Sets PRINT cursor position
ATAN	7	Function	Arc tangent – gives radians
AUTO	3	Statement	Auto line numbering
BAUD	9	Statement	Sets Baud rate of serial ports
BEEP	8	Statement	Causes QL to make a sound
BEEPING	8	Function	True if sound being produced
BLOCK	8	Statement	Draws rectangle of pixels
BORDER	8	Statement	Draws border to a window
BREAK	3	Keypress	Stops execution (CTRL SPACE)
CALL	4	Statement	Executes machine code program
CHR$	6	Function	Character corresponding to ASCII
CIRCLE	8	Statement	Draws circles and ellipses
CLEAR	3	Statement	Clears all variable values
CLOSE	9	Statement	Closes files and channels
CLS	4	Statement	Clears whole or part window
CODE	6	Function	Gives ASCII for character
CONTINUE	5	Statement	Continues after BREAK
COPY	9	Statement	Copies data between devices
COS	7	Function	Cosine (radians)
COT	7	Function	Cotangent (radians)
CSIZE	8	Statement	Changes character size
CURSOR	8	Statement	Sets PRINT position in graphics

Keyword	*Chapter*	*Type*	*Description*
DATA	4	Statement	Holds blocks of data
DATE	4	Function	Date as floating point number
DAY$	4	Function	Day in letters
DEF FN	10	Statement	Defines new function
DEF PROC	10	Statement	Defines a procedure
DEG	7	Function	Changes radians into degrees
DELETE	9	Statement	Deletes files from microdrives
DIM	6	Statement	Sets dimension of arrays
DIMN	6	Function	Gives current array dimension
DIR	9	Statement	Displays microdrive directory
DIV	7	Operator	Integer divide
DLINE	3	Statement	Deletes range of program lines
EDIT	3	Statement	Edits a program line
ELSE	5	Statement	Part of IF THEN statement
END DEF	10	Statement	Ends procedure and DEF FN blocks
END FOR	5	Statement	Ends FOR block
END IF	5	Statement	Ends IF block
END REP	5	Statement	Ends REPEAT loop
END SEL	5	Statement	Ends SELECT block
EXEC	10	Statement	Executes multitasking routines
EXIT	5	Statement	Ends a loop
EXP	7	Function	Raises *e* to a power
FILL	8	Statement	Fills a convex shape
FILL$	6	Function	Gives repetition of a character
FLASH	8	Statement	Creates flashing characters
FOR	5	Statement	FOR loop
FORMAT	9	Statement	Formats a microdrive tape
GOSUB	10	Statement	Calls a subroutine
GOTO	6	Statement	Executes given line number
IF	5	Statement	Conditional statement
INK	8	Statement	Sets colour of screen output
INKEY$	4	Function	Inputs single keyboard character
INPUT	4	Statement	Inputs values from keyboard
INSTR	6	Operator	Checks for given substring
INT	7	Function	Gives integer less than argument
KEYROW	4	Function	Gives column of a keypressing
LBYTES	9	Function	Loads binary data
LEN	6	Function	Gives length of string
LET	3	Statement	Optional assignment statement
LINE	8	Statement	Draws lines
LIST	3	Statement	Lists current program
LN	7	Function	Natural logarithm (base *e*)
LOAD	9	Statement	Loads program from microdrive
LOC	10	Statement	Makes variables local
LOG10	7	Function	Logarithm to base 10
LRUN	9	Statement	Loads and runs program file

Keyword	*Chapter*	*Type*	*Description*
MERGE	9	Statement	Merges program file into memory
MOD	7	Operator	Remainder on integer division
MODE	8	Statement	Changes display resolution mode
MOVE	8	Statement	Move turtle across screen
MRUN	9	Statement	Merge and run program from file
NET	9	Statement	Sets network station number
NEW	3	Statement	Wipes program from memory
NEXT	5	Statement	End of FOR loop
NOT	5	Operator	Logical complement
ON GOSUB	10	Statement	Switch to subroutines
ON GOTO	10	Statement	Switch to line numbers
OPEN	8 & 9	Statement	Opens device channels
OPEN_IN	9	Statement	Opens file for reading
OPEN_NEW	9	Statement	Opens new file for writing
OR	5	Operator	Inclusive Boolean OR
OVER	8	Statement	Overprints display
PAN	8	Statement	Moves window contents horizontally
PAPER	8	Statement	Sets background colour
PAUSE	3	Statement	Causes execution to pause
PEEK	4	Function	Gives memory contents
PENDOWN	8	Statement	Commands turtle to leave trail
PENUP	8	Statement	Stops turtle trail
PI	7	Function	Gives value of pi
POINT	8	Statement	Draws a graphics point
POKE	4	Statement	Sets memory locations
PRINT	4	Statement	Prints characters on screen
RAD	7	Function	Converts degrees to radians
RANDOMISE	7	Statement	Sets random number generator seed
READ	4	Statement	Reads from block of data
RECOL	8	Statement	Recolours all pixels
REM	3	Statement	Allows remarks in program
RENUM	3	Statement	Renumbers program lines
REP	5	Statement	Repeat loop header
RESTORE	4	Statement	Restores pointer in DATA block
RET	10	Statement	Returns from DEF FN, PROC, GOSUB
RETRY	5	Statement	Retries last statement executed
RND	7	Function	Gives a random number
RUN	3	Statement	Runs current program
SAVE	9	Statement	Saves program on microdrive
SBYTES	9	Statement	Saves binary data on microdrive
SCALE	8	Statement	Sets graphics scale
SCROLL	8	Statement	Moves window contents vertically
SDATE	4	Statement	Sets real time clock
SEL	5	Statement	Selects alternative actions
SEXEC	9	Statement	Saves for multitasking EXEC
SIN	7	Function	Sine (radians)

Keyword	*Chapter*	*Type*	*Description*
SQRT	7	Function	Square root
STOP	5	Statement	Stops program execution
STRIP	8	Statement	Sets background to characters
TAN	7	Function	Tangent (radians)
THEN	5	Statement	IF THEN
TURN	8	Statement	Changes direction of turtle
TURNTO	8	Statement	Sets heading of turtle
UNDER	8	Statement	Underlines characters
WINDOW	8	Statement	Sets characteristics of windows
XOR	5	Operator	Exclusive OR Boolean

Index

ABS, 106
ADATE, 58
addition, 30
address, 61
ALT, 4
AND, 71
ARC, 130
argument, 58
arrays, 93
array variable, 19, 93
arrow keys, 4
ASCII, 91
assignment, 11
ATAN ACOT, 108
AT, 50
AUTO, 37

backslash, 49
backup, 144
BASIC, 7
BAUD, 142
BEEPING, 134
BEEP, 5, 134
binary operator, 109
BLOCK, 115
Boolean, 69
Boolean operator, 71
BORDER, 113, 118, 120
bracket, 31
BREAK, 38
bubble sort, 24, 96
bug, 7

CALL, 61
CAPSLOCK, 4
channel, 118
CHR$, 92
CIRCLE, 129
CLEAR, 29
clock, 58
CLOSE, 120, 146
CLS, 50, 123
CODE, 91
coercion, 34, 101
colon, 43
colour, 115, 117
command window, 3
concatenate, 33
CON, 140
CONTINUE, 42, 80
co-ordinate system, 115
COPY, 144
CSIZE, 125
CTRL, 4
CURSOR, 3, 128

DATA, 54
data pointer, 55
data type, 34
DATE$, 58
DATE, 59
DAY$, 58
decision, 16, 67
default, 37
DEF FN, 157
DEF PROC, 150
DEG, 107
DELETE, 142
devices, 140
DIM, 94
DIMN, 95
DIR, 139
division, 30
DIV, 106
DLINE, 39

editing, 28
EDIT, 44
ELSE, 84
END DEF, 151

END FOR, 81
END IF, 83
END REP, 79
END SEL, 88
ENTER, 4
ESC, 4
evaluation order, 31
exclamation mark, 49
EXIT, 77
exponent form, 35
EXP, 108
expression, 31

files, 145
FILL, 131
FILL$, 100
FLASH, 124
floating point, 34
flowchart, 15
FORMAT, 76, 138
FOR TO STEP NEXT, 64
function, 57
function keys, 2

global, 153
GOSUB, 156
GOTO, 49
graphics co-ord system, 126
graphics cursor, 127
graphics, 112
greater than, 24

high resolution mode, 112

identifier, 10
IF THEN, 67
indices, 19
INKEY$, 57
INK, 113, 118
INPUT, 51
input/output, 9, 46
INSTR, 102
integer, 11, 34
INT, 106

KEYROW, 58
keyword, 12

labelling, 9
language, 1
LBYTES, 144
LEN, 102
LET, 11
LINE, 129
LIST, 30, 39
literals, 12
LOAD, 77, 142
LOC, 153
logical statement, 69
LOGS, 108
loop counter, 21
loop identifier, 77
LOOP, 18, 21, 41, 63
low resolution mode, 112
LRUN, 143

MDV1, 137
MDV2, 137
MERGE, 143
Microdrive, 137
MISTAKE, 142
MODE, 113
MOD, 107
monitor, 2
MOVE, 133
MPU, 8
MRUN, 143
multiplication, 30

nesting, 82
NET, 140
NEW, 29
NOT, 72
numeric function, 105
numeric operator, 71
numeric, 11

ON GOSUB, 156
ON GOTO, 86
ON, 88
OPEN_IN, 145
OPEN_NEW, 145
OPEN, 119
origin, 126
OR, 72
output window, 15
OVER, 125

PAN, 122
PAPER, 113, 118
parameter, 6
PAUSE, 42
PEEK, 60
PENDOWN, 133
PENUP, 133
PI, 107
pixel, 112
pixel drawing system, 114

POINT, 127
POKE, 60
powers, 110
PRINTER, 140, 142
PRINT list, 48
PRINT, 13
procedure, 150
program, 1

RAD, 107
random numbers, 108
READ, 55
RECOL, 124
record, 145
relation, 9, 72
relative mode, 126
REMAINDER, 88
REMarks, 40
RENUMber, 38
REP, 77
reset, 2
RESTORE, 55
RET, 156
RETRY, 80
RND, 73
RUN, 29, 40

SAVE, 76, 141
SBYTES, 144
SCALE, 131
SCROLL, 3, 121
SCR, 140
SDATE, 58
sector, 139
SEL, 87
semicolon, 48
separator, 46
sequential execution, 27
SER, 140
SEXEC, 144
SHIFT, 3
slicing strings, 98
software, 7
sorting, 9, 23
sound, 134
SPACE BAR, 3
SQRT, 105
statement, 1, 7
stipple, 117
stone sort, 24
STOP, 80
storage, 9
stored program, 27
string ordering, 96
string, 11, 33, 91
STRIP, 125
subroutine, 156
subscripts, 19
substring, 99
subtraction, 30
SuperBASIC, 1
syntax, 28

TO, 38
transparent, 8
trig functions, 107
TURN, 133
TURNTO, 133
turtle graphics, 133

unary minus, 34
UNDER, 126

variable, 9

WINDOW, 13, 113, 118, 120

XOR, 72

UNLOCK THE QL'S POTENTIAL WITH SUPERBASIC!

To get the best out of the QL you need to master the QL's special dialect of BASIC—one of the most sophisticated and powerful versions ever devised.

This book takes you through the whole repertoire of QL SuperBASIC in easy stages, so that you quickly become a competent and effective programmer. Many programs are provided as examples, setting out the principles of structured programming and the applications which are possible with the QL.

The book is suitable for both beginners and most experienced users, and will continue to be a useful source of reference long after you have learnt the language.

The Author

Dr A. A. Berk is a well-known expert on microcomputers, and has written many technical manuals, articles for the press and previous books. He has also spent a number of years teaching programming languages, including BASIC, and has designed a number of educational machines and projects.

More books for QL users

QL COMPUTING
Ian Sinclair
0 246 12595 0

QL GAMESMASTER
Kay Ewbank, Mike James and S. M. Gee
0 00 383066 7

PROFESSIONAL AND BUSINESS USES OF THE QL
Colin Lewis
0 00 383005 5

Front cover illustration by Tony Roberts

GRANADA PUBLISHING
Printed in Great Britain 0 246 12596 9

Chiquinquirá, 28 - (entr. Cecut)
Tl. 764 53 95 - 28033-MADRID