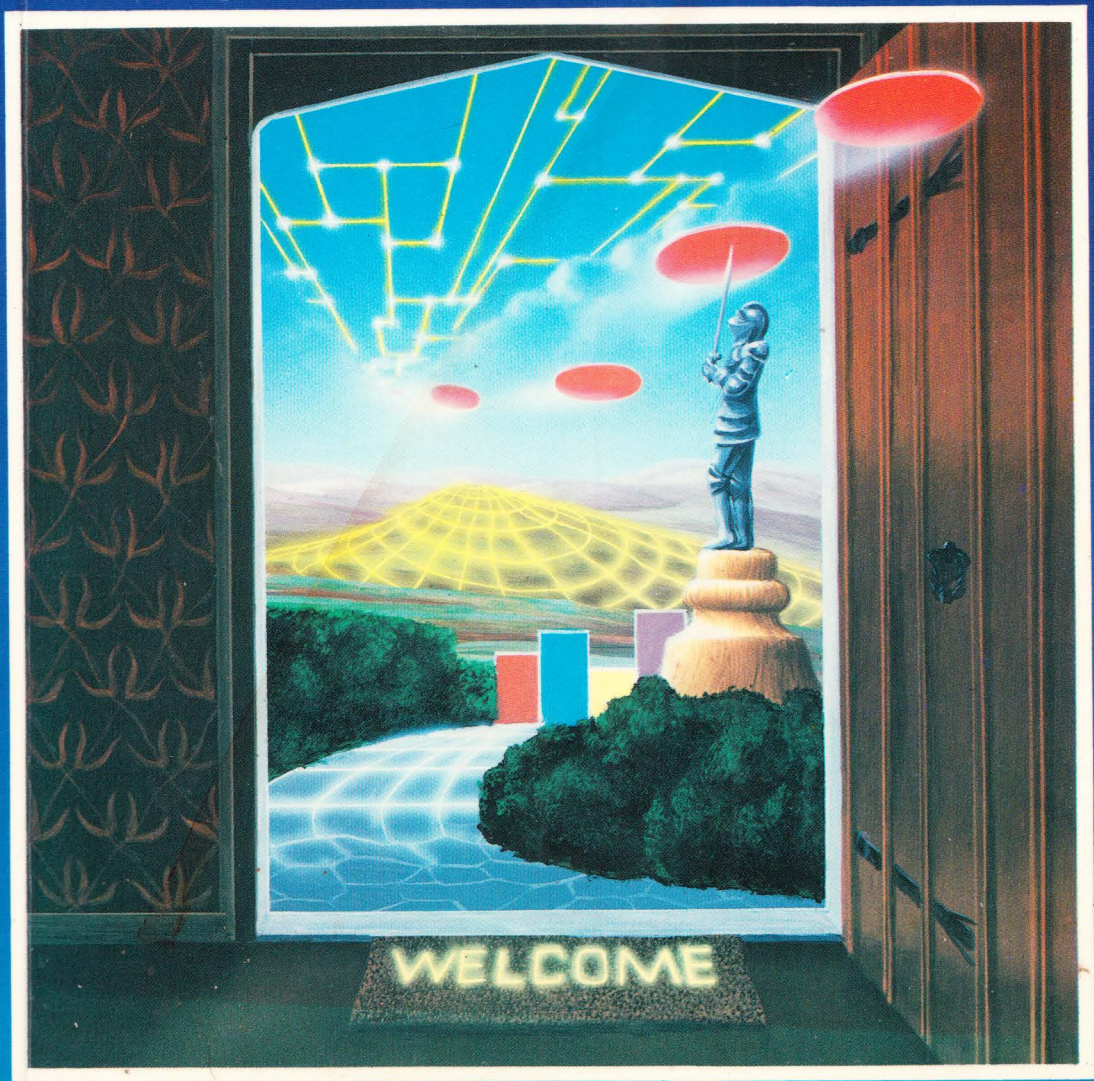
The Commodore C16/plus 4 Companion

A beginners guide

Brian Lloyd



The Commodore C16/plus4 Companion

A beginners guide

Brian Lloyd

First published 1984 by:
Sunshine Books (an imprint of Scot Press Ltd.)
12–13 Little Newport Street
London WC2H 7PP

British Library Cataloguing in Publication Data
Lloyd, Brian
The Commodore C16/Plus 4 Companion.
1. Commodore C16 (Computer)
2. Commodore Plus 4 (Computer)
I. Title
001.64'04 QA76.8.C64

ISBN 0–946408–64–5

Cover design by Grad Graphic Design Ltd.
Cover illustration by Stuart Hughes.
Typeset and printed in England by Commercial Colour Press, London E7.

CONTENTS

Contents in detail

Introduction

This book has been written for the complete newcomer to the world of computers, and this has been kept in mind throughout. This will not prevent the experienced programmer from getting to know the version of BASIC which is used in the Commodore Plus/4 and C16 computers, as each command is fully explained. You will be shown how to use your computer, and how to make it do what you want it to do. Everything is fully explained, from the simple facts like how to plug everything in, right up to the more sophisticated programming techniques of file handling and PEEKing and POKEing.

Although everything is explained fully, I have tried to avoid long, unnecessary explanations of very simple facts. Instead, the book is designed in such a way that you can get down to writing your own programs from a very early stage. Commands are introduced as you need to know them, so that you do not become confused by the complicated commands before you even know how to use the keyboard.

You are advised to read through the book line by line, and not skip a few pages because it looks boring. This is difficult to do, because everybody is naturally eager to get on and try something new. Unfortunately, however, you can easily become lost in this way, so it is best to work through the book slowly and methodically if you really want to become a good computer programmer. If you do not understand something fully the first time you read it, then go back and read it again.

The most important thing is to experiment, for this is undoubtedly the best way to learn. Try out everything you learn to see how it works (and also why it doesn't!). Do not be afraid of doing any harm to the computer, for unless you set about it with a hammer there's not much harm you can really do.

Finally, before you start reading through the book, you may like to know that the main language of your computer, and the one which this book is designed to teach you, is called BASIC. This stands for Beginner's All Purpose Symbolic Instruction Code, and although this may seem a little complicated, you should be reassured by the fact the that the first word is 'Beginner's'. The fact that BASIC is designed to be a beginner's language should not put you off either, because the language is extremely powerful.

Acknowledgements

My thanks to the staff of Commodore Business Machines (UK) Ltd, especially to Gail Wellington, Phil Gosling, and the technical staff, without whose patience and help this book could not have been written.

CHAPTER 1

Getting Started

Connecting up

The first, and most important, step in learning how to use your new Commodore Plus/4 or C16 is to learn how to connect up your computer. Your computer needs to be connected to the power supply and a television, and later to a cassette recorder, and possibly a disk drive and printer, all of which we will deal with in due course.

The first thing to connect up is the power supply. You will have to fit a suitable mains plug. The din plug on the end of the power supply cable should be plugged into the socket marked power (this is at the left of the Plus/4 as you look at it from the back, and on the right of the C16), after making sure that the on-off switch on the right of your computer is in the off position.

You will also have a TV lead with your computer. On one end of this lead is a phono type plug, and this plugs into the socket marked RF (this is on the left of the Plus/4, and at the back of the C16). The plug on the other end of the lead goes into the back of your television.

There is no need to connect anything else at this stage, any cassette recorders, disk drives or any other add-ons which you may have can be connected as you learn how to use them.

Now, the big moment! Turn on your television, switch on your computer (a red light should appear), and, unless you are very lucky, you will have a picture on your television screen which looks exactly as if there was no aerial connected, and your ears will be assaulted by a nasty hissing noise. All you now need to do is to tune your television in to your computer (somewhere around channel 36). When you have done this you will either be confronted by a message saying either

COMMODORE BASIC V3.5 60671 BYTES FREE

READY

(this is for the Plus/4) or

COMMODORE BASIC V3.5 12277 BYTES FREE

READY

(this is for the C16).

This is the start-up message and tells you that your computer understands Commodore BASIC Version 3.5, and also tells you how many bytes of memory you have available (one byte can hold one character, ie. it would need five bytes to hold the word HELLO as this is made up of five characters). The word READY tells you that your computer is waiting for you to tell it to do something. Beneath READY you will see a flashing square, called a cursor. The cursor is there to tell you where on the screen the next character that you type on the keyboard will appear.

If you have a Plus/4 then you will also be told that the 3-PLUS-1 ROM is fitted, and which key to press to use it. You should refer to your 3-PLUS-1 manual for instructions on how to use this excellent facility.

The keyboard

Now to the keyboard. At first this looks to be a very complicated device, but it does not take very long to get used to it. Those of you who are familiar with typewriter keyboards will probably notice that the letter keys of your computer are arranged in the standard qwerty layout (the layout is called qwerty because the first six letters on the keyboard are q, w, e, r, t and y). However, the keys on your computer have many more symbols on them than those on a typewriter, and there are also several extra keys.

Normally when you type on a typewriter the letters appear on a piece of paper and are lower case (or small) letters. However, when you type on a computer the letters appear on the television screen and are upper case (or capital) letters. If you type

HELLO EVERYBODY

you will see the letters appearing on the screen in upper case, with the cursor moving along from left to right as you type.

Lower case letters

Your computer can, however, produce lower case letters. If you hold down one of the keys marked 'SHIFT' and at the same time press the Commodore key (the one at the bottom left of the keyboard with the Commodore symbol on it) you will see all the letters on the screen change to lower case. If you now type

hello everybody

you will see the letters appearing on the screen in lower case.

When you want to get upper case letters on a typewriter you have to hold down the shift key while you type the letter which you require. The same goes for your computer when it is in lower case mode. For instance, in order to type

Hello Fred

you must first make sure that you are in lower case mode, and then hold down the shift key while pressing the H key, then let go of the shift and type ello, press the space bar, then hold down the shift key while pressing the F key, and then type red.

When you come to enter commands you will find that they can be entered in upper and lower case letters. If, however, you are in lower case mode and you shift a letter to get upper case and type a command, then the command will be ignored.

Shifted symbols

The shift key is also used to obtain the characters above the numbers. For instance, to obtain a dollar symbol you must hold down the shift key and then press the 4 key. The same is true for the comma, full stop, '/', ':' and ';' keys, each of which has two symbols on the key top. In order to obtain any of the uppermost characters you must hold down the shift key and press the key which has the symbol you require.

Clearing the screen

By now the screen will probably be getting quite full. Fortunately for us, it is very easy to clear the screen of all its contents. If you press the key marked CLEAR/HOME the cursor will jump to the top left-hand corner of the screen. If you hold down the shift key and press the CLEAR/HOME key then the screen will be wiped clear and the cursor will appear in the top left-hand corner of the screen, ready for you to start again.

The cursor control keys

You will probably have noticed the four arrow keys. These keys are called the cursor control keys, and pressing one of these keys moves the cursor along in the direction in which that key is pointing. These keys allow you to choose where you want the characters that you type to appear.

The INST/DEL key

In the top righthand corner of the keyboard is a key marked INST/DEL.

This key allows you to DELete characters from the screen, and to INSerT spaces between characters. For instance, make sure that your computer is in upper case mode (remember — shift and the Commodore key allow you to switch between upper and lower case) and type in the following

JACK WALKED THEE RODE

Now, the first thing to do is to correct the spelling of RODE. Move the cursor so that it is over the letter O (do this by using the cursor control keys) and then type OAD. These new letters will replace the existing ones and you will be left with the words

JACK WALKED THEE ROAD

on the screen. The next step is to get rid of the extra E in THE. To do this you must position the cursor on the space after the second E and press the INST/DEL key. The extra E will be deleted, like this

JACK WALKED THE ROAD

However, this still doesn't make sense — we need to insert the word ALONG between WALKED and THE. To do this you must first position the cursor on the T of THE. If you now hold down the shift key and press the INST/DEL key six times (still holding down the shift key) you will see the words THE ROAD moved right by six spaces. You can now release the shift key and type ALONG (remember to type a space), and you are left with the sentence

JACK WALKED ALONG THE ROAD

The SHIFT LOCK

Above the lefthand SHIFT key is a SHIFT LOCK key. This key does as its name suggests — it locks the SHIFT on, meaning that you do not need to hold down the SHIFT key all the time in order to get shifted characters. If you press the SHIFT LOCK once you will find that it clicks into position about half way down. The SHIFT LOCK is now on. Pressing the SHIFT LOCK a second time releases the shift.

The CONTROL key

Another key with a special function is the CONTROL key. The Plus/4 computer has two control keys, one at either end of the second row of keys. The C16 only has one CONTROL key, but this is no real disadvantage. If

you look at the number keys from 1 to 8 you will see that each one has two colours shown on the front of it. By holding down the CONTROL key and pressing one of these keys you can change the cursor colour (and the colour of any text that you then type) to the uppermost colour on that key.

The Commodore key

The Commodore key can be used to select the colours written on the bottoms of the number keys. For example, if you press the Commodore key and the 4 key at the same time then the cursor colour will change to pink.

Reverse on and off

The CONTROL key can also be used to obtain reversed characters. In other words, if, for example, the normal text colour is black on white, then when the text is reversed it becomes white on black. You may understand better if you try typing a few characters, and then hold down the CONTROL key and press the 9 key (the one with ReVerSe ON written on the front of it). Let go of both keys and type in a few characters. You will then see what reversed characters look like. To return to normal just hold down the CONTROL key and press the 0 key (the one with ReVerSe OFF written on it).

Flashing characters

One other use of the CONTROL key is to obtain flashing characters. If you hold down the CONTROL key and press the comma key (this has 'Flash On' written on the front of it), then any characters which you type will flash on the screen. Pressing CONTROL and the full stop key (which has 'Flash Off' written on it) makes any further characters that you type in appear on the screen as normal, but any characters that are already flashing will continue to flash.

Graphic symbols

You will probably have noticed that on the fronts of many of the keys are some strange symbols make up of lines and curves. These are graphics symbols and are obtained with the aid of the SHIFT and Commodore keys. Make sure that your computer is in upper case mode, then hold down the SHIFT key (or press the SHIFT LOCK) and try pressing a few of the keys with graphics symbols marked on them. You should notice that the SHIFT

key allows you to obtain the right-hand graphics symbols. If you now release the SHIFT (or SHIFT LOCK) key and instead hold down the Commodore key while pressing keys which have graphics symbols on them you will be able to obtain the left-hand graphics symbols.

If you now switch to lower case mode you will see that some of the graphics symbols on the screen will change to capital letters. This is because when you are in lower case mode, SHIFTing a letter key will produce an upper case letter instead of a graphic symbol. In other words, when you are in lower case mode you lose all the right hand graphic symbols on the letter keys, and if you SHIFT any of these keys you get upper case letters.

The RETURN key

At some time during your experiments with the keyboard you may have pressed the RETURN key, in which case you may have received the message '? SYNTAX ERROR'. If this happens when you are experimenting with the keyboard then do not worry. The RETURN key tells the computer to carry out any instructions which you have typed in, so if you type in the word 'HELLO' and then press the RETURN key you will receive the message 'SYNTAX ERROR', because your computer does not understand the word 'HELLO'.

There are now only six keys which you do not know how to use — the four function keys (F1 to F8), the ESC key, and the RUN/STOP key. These will be explained later.

The RESET button

On the righthand side of your computer is a small button marked 'RESET'. Pressing this button returns the computer to a cold start (which means that the contents of the memory are lost), and the normal start-up message is displayed. However, if you hold down the RUN/STOP key and press this button, the screen will clear and the message 'MONITOR' followed by a series of letters and numbers below. If you now type X and press the RETURN key then everything will be back to normal, and whatever program you have in memory will still be there (this is a warm start).

Summary

Here is a brief summary of what the special keys are used for.

SHIFT: To obtain shifted symbols (such as $, &, [), to obtain the righthand graphics symbols, and used in conjunction with the Commodore key to switch between upper and lower case letters.

SHIFT LOCK: Holds the SHIFT on.

COMMODORE: To obtain the bottom colours on the number keys (1 to 8), to obtain the lefthand graphics symbols, and used in conjunction with the shift key to switch between upper and lower case letters.

CLEAR/HOME: Used to move the cursor to its home position (the top left-hand corner of the screen). When shifted, it is used to clear the screen.

INST/DEL: Used to delete characters. When shifted it inserts spaces between characters.

CONTROL: Used to obtain the top colours on the number keys (1 to 8) and in conjunction with the 9 and 0 keys to turn on and off the reversed characters. Also used in conjunction with the comma and full stop keys to turn on and off the flashing characters.

ARROW KEYS: Used to move the cursor around the screen.

CHAPTER 2

The Print Statement

By now you should be quite used to the keyboard and know roughly where everything is, so you are ready to start making your computer work for you.

One of the basic tasks which a computer can carry out is performing calculations. In order to perform calculations you need to use the PRINT command.

Let's say, for instance, that you want to work out the answer to 9 + 7. To do this you must type in this command

```
PRINT 9+7
```

and then press the RETURN key. Pressing this key tells the computer to carry out the command which you have just typed in, so in this case the computer will carry out the calculation 9 + 7 and display the answer on the screen.

If you receive an error message instead of the answer then you will have made a typing mistake. In this case you just have to move the cursor up to the place where you made the mistake, correct it (in the same way as you corrected the sentence in the last chapter) and press the RETURN key.

When you entered the above command you told the computer to PRINT (or display on the screen) the answer to the sum 9 + 7. Of course, your computer can carry out other calculations besides addition. Try the following (remember to press the RETURN key after each command)

```
PRINT 32-17
PRINT 9*54
PRINT 99/11
PRINT 1564+1928
```

You may have worked out from these calculations that the * sign means 'multiplied by' and the / sign means 'divided by'. This form of notation is common to nearly all computers. Try some calculations of your own and see what results you get.

If you have tried some multiple calculations (such as 92 + 8/4) then you

may have thought that the computer had made a mistake with its answer. Try this calculation

```
PRINT 6+4/2
```

You may be surprised to see that the answer is 8 and not 5. This is because computers always give certain calculations priority over others. In the case of your computer (and most others), multiplication and division are always carried out before subtraction and addition. This means that the computer works out the above sum as 6 + (the answer to 4/2), which, of course, is 8. To get the answer 5 we must do this

```
PRINT (6+4)/2
```

Putting brackets around the '6 + 4' tells the computer to work out this part of the calculation first, so the sum is worked out as (the answer to 6 + 4) divided by two, which is 5.

You may have noticed that there is an up arrow on the 0 key. This symbol means 'to the power of', so if you type

```
PRINT 3↑4
```

then the answer 81 will appear on your screen, because 3 to the power of 4 is 81 (3 times 3 times 3 times 3 is 81).

Exponentiation, as it is called, is worked out before any other calculation, so the order of priority is

Exponentiation, multiplication or division, addition or subtraction

If, for instance,the computer has to work out the answer to 5 + 7 – 2, then it will work out the calculation in the order in which it has been written, because addition and subtraction have equal priority, as have multiplication and division.

It is also possible to work in negative numbers. All you have to do is to put a minus sign in front of a number to indicate that it is negative. For instance

```
PRINT -2+7
```

will give the answer 5.

Summary

The PRINT statement must be used in order to carry out calculations, and

takes the format PRINT [calculation]. Calculations are carried out in the following order

Exponentiation
Multiplication or division
Addition or subtraction

If a calculation contains, for example, a subtraction followed by an addition, then the subtraction is carried out first, because the two types of calculation have equal priority.

Brackets can be put around a part of a calculation in order to make the computer carry out that part of the calculation first, regardless of what it is.

Printing messages

One of the other functions which the PRINT statement can do is to display messages on the screen. If you type

PRINT"HELLO, HOW ARE YOU?"

(and, of course, press the RETURN key) you will see the message HELLO, HOW ARE YOU? appear on the screen.

The PRINT statement will display any message which you enclose in quotation marks (") on the screen. The quotation marks tell the computer that what is enclosed within them is not a calculation and is exactly what you want on the screen.

Changing the text colour

It can be useful to be able to display a message in several different colours. To do this we must include colour changes in the PRINT statements which we use to display these messages. This is very easy to do — you simply have to type in the PRINT statement as normal and then, where you want to change the colour you hold down control or Commodore (depending on which colour you want) and then the number key which has the colour you require written on it. For instance, if you wanted to display the message HELLO FRED with the word HELLO in red and the word FRED in blue, then you would type this in (where you see something enclosed in square brackets you should carry out these instructions, but do not type the square brackets or what is enclosed in them)

PRINT"[*hold down the control key and press the 3 key, then release both*

keys]HELLO[*hold down the CONTROL key and press the 7 key then release both*]FRED"

When you press RETURN you will see this colourful message appear on the screen. As you can see, when you change the colour in the PRINT statement, a reverse-field character appears. This is to remind you where you changed colour, and what colour you changed to.

The abbreviation for print

One useful thing to know is that instead of typing PRINT each time you can instead use the ? symbol. The computer will read the ? as PRINT and carry on as if you had used the full command. For example, ?"HELLO" will give exactly the same results as PRINT"HELLO". This saves a lot of typing, so try to remember it.

You have now taken the first steps in learning how to become a good computer programmer. You already know how to make your computer work out the answers to calculations (using the PRINT statement), and also know how to display messages on the screen in different colours. You know how to use the keyboard and that the computer will not carry out any command until you press the RETURN key.

Summary

Anything enclosed in quotation marks following the PRINT statement will be displayed on the screen exactly as it appears between the quotation marks.

Changing the text colour in a PRINT statement is done in the same way as changing the text colour normally. A reverse-field character is displayed to indicate where you changed the colour, and what you changed it to.

The PRINT statement may be abbreviated to ?.

It would be a good idea if you took half an hour or so to experiment with what you know so far and make sure that you fully understand everything. This will save you from having to keep going back through the book every five minutes, which is important because soon you will be writing your very first BASIC program.

CHAPTER 3

Your First Program

Line numbers

You will probably have heard of computer programs (the American spelling is used for computer programs), and have probably wondered exactly what they are. A computer program is a series of commands which are carried out in a set order by the computer. Until now you have only been telling the computer to carry out each command as you type it in — this is called COMMAND MODE, and once the computer carries out a command in command mode it is forgotten. This is of no use whatsoever when you are writing a program, because you do not want the commands to be carried out immediately. To overcome this problem, when writing programs we use line numbers. This also allows us to carry out a program again and again, as many times as we need to.

Line numbers are used to make a computer remember each command that you give it and to tell it in which order you want the commands to be carried out in. Most people like to have their line numbers in increments of ten (i.e. 10, 20, 30), as this allows them plenty of room to add extra lines in between, as you will see in a moment.

So here it is, your first program. You will notice that each command is on a separate line with a line number. Try typing in this program exactly as it appears (remembering to press RETURN at the end of each line)

```
10 SCNCLR
20 PRINT "HELLO!!!"
30 PRINT "THIS IS YOUR FIRST COMPUTER PROGRAM?"
40 PRINT "ITS NOT EXACTLY IMPRESSIVE,"
50 PRINT "BUT THE PROGRAMS WILL IMPROVE!"
```

Well, not a lot happened did it? This is because all we have done is typed in a program which the computer will remember until we tell it to forget it. To make the program work you will need to type the command RUN (and press the RETURN key, of course). As soon as you do this the computer will carry out the program.

You may get an error message at some time, in which case the computer

will tell you which line has the mistake in it. If this happens you will need to re-type that line.

SCNCLR

You may have noticed a new command on line 10. This command tells the computer to clear the screen, in exactly the same way as when you press SHIFT and the CLEAR/HOME key. All the rest of the program is made up of PRINT statements, so you should be able to work out what is happening. An alternative to the SCNCLR command is to PRINT a heart symbol, which is obtained by holding down the SHIFT key and pressing the CLEAR/HOME key.

So, the program is in memory, and it has been carried out, but now what? Well, it is possible to see the program you typed in, simply by typing LIST (not forgetting to press RETURN, of course). This command will make the program appear on the screen line by line (although this does happen quite quickly).

One thing which you may have noticed is that when the program is RUN, each PRINT statement displays its message on a separate line. It is possible, however, to have one PRINT statement continue where another left off. This may sound confusing, so to see how this works add these lines to the program

```
60 PRINT"3232 + 5674 = ";
70 PRINT3232 + 5674
```

When you now RUN the program you will see the same message as before, but at the end will be a line saying '3232 + 5674 = 8906'.

Line 60 of the program tells the computer to display the characters '3232 + 5674 = ', and line 70 tells the computer to work out the answer to the calculation 3232 + 5674 and display it on the screen. The important question is, why did the second PRINT statement display the answer on the same line as the first PRINT statement displayed the calculation? If you look at the end of line 60 you will see a semicolon (;), and it is this which tells the computer not to start the next lot of PRINTing on a new line, but to carry on where it left off.

You may remember that at the beginning of this chapter I mentioned that most people like to number their program lines in steps of ten, allowing them to add more lines in between at a later date. To see what is meant by this, type in this line and then type LIST.

```
55 PRINT"A CALCULATION: – "
```

When the program appears on the screen you will see that the new line (line

55), has been inserted between lines 50 and 60. Try RUNning the program and see what effect this line has.

If you made a mistake when typing in this program you would have had to type in the offending line again. The reason why this had the effect of correcting the mistake is that if you give a line the same line number as one which already exists then the computer will replace the old line with the new one. This may seem confusing at first, so to illustrate this point type in this line.

```
55 PRINT"THIS LINE HAS CHANGED!"
```

When you type LIST you will see that the old line 55 has been replaced by a new one, the one which you just typed in.

Deleting a line

It is also possible to delete a line from a program if it is not needed. To do this you simply type the number of the line which you want removed and press the RETURN key. For instance, type the number 55 and then press RETURN — line 55 will be removed from the program (type LIST to make sure).

Multi-statement lines

So far we have had only one command to each line. However, your computer does allow you to have more than one command on each line, in fact you can have as many commands as you like as long as the total length of the program line is not more than eighty characters (or two lines on the screen). There are some exceptions, but these will be explained as we come across them. Try changing line 10 to this

```
10 SCNCLR:PRINT"THIS IS A MULTI-STATEMENT LINE"
```

Line 10 now has two commands, the SCNCLR command and a PRINT statement, separated by a colon (:). All you have to do to have more than one command on a line is to separate each command with a colon.

Summary

In order to make the computer remember a series of instructions we must give those instructions line numbers. This allows us to set which order we want the instructions to be carried out in, and also allows us to carry out the commands as often as we like.

Line numbers are usually in increments of ten to allow extra lines to be added at a later date. This is not essential but it is helpful when you want to add to a program.

Program lines may easily be deleted simply by typing the line number and pressing RETURN.

It is possible to display a program on the screen by typing LIST. A program can be carried out by typing RUN.

More than one statement may be included on a line. Each statement must be separated from the last by a colon (:).

It may be a good idea if you experimented with this program, or even tried writing one of your own (in this case you will have to type NEW to get rid of the old program). This will help you to understand one of the most important parts of programming in BASIC.

Numeric variables

Variables are extremely important to the BASIC programmer, and without them it would be very difficult to write a program which would be of any real use.

A variable is a value which can be changed and there are three types of variables. We shall look at numeric variables first of all.

Letters are used to represent these values, so for instance we can tell the computer that we want the variable A to represent the number 34. Try typing in this short program (you must type NEW and RETURN first, however, to get rid of any programs which may be in memory)

```
10 SCNCLR
20 A = 34:PRINT A
```

When you RUN this program you will see the screen clear and the number 34 appear on the screen. This is because we have told the computer that we want the variable A to represent the number 34, and whenever we refer to the variable A (as we did when we told it to PRINT A), then it should carry on as if we had really referred to the number 34. You can, of course, tell the computer to make the variable A represent any number.

Because a variable can be used to represent a number we can treat them just like numbers — we can add, divide or do anything that we can normally do with numbers. Try adding the following lines to your program to illustrate this

```
30 B = 52:PRINT B
40 PRINT"34 + 52 = ";A + B
50 PRINT"34*52 = ";A*B
60 C = B – A
70 PRINT C
```

Here is an explanation of what the program is doing.

Line 10: Clears the screen
Line 20: Remember that the variable A represents the number 34, then display the number represented by A on the screen.
Line 30: Remember that the variable B represents the number 52, then display the number represented by B on the screen.
Line 40: Display the messge 34 + 32 = , then look to see what the variables A and B represent and add them together before displaying the answer on the same line.
Line 50: Display the message 34*32 = , then look to see what the variables A and B represent and multiply them before displaying the answer on the same line.
Line 60: Look to see what the variables A and B represent, then subtract A from B and remember that the variable C represents the answer.
Line 70: Look to see what the variable C represents and display that number on the screen.

A variable can be any letter, or virtually any combination of letters and numbers, so B, HELLO, FRED, A3, ZT99 and NUMBER9 can all be used as variables. However, all variables *must* start with a letter, so A9 can be used as a variable, but 9A cannot.

A variable cannot be used if a command is in the variable, so PRINTER cannot be used (because the PRINT statement is in the command), nor can TRUNDLE (RUN) or BLISTER (LIST).

A variable can be of any length (as long as it fits onto a program line), but the computer only recognises the first two letters of the variable. This means that the variables HELLO and HEXAGON are both read by the computer as HE, so are therefore both the same variable.

The computer sets all variables to zero until you use them, so if you want to you can say A = B, even if you have not used the variable B before, the computer will just set the variable A to zero.

Finally, we have been assigning values to variables simply by saying, for instance, A = 1. There is a BASIC command, called LET, which can be used to assign values to variables. However, it is not necessary to use this command because the computer always reads the command LET A = 1 as A = 1, which is exactly the way we have been assigning values to variables. If you ever see the LET command in a program then you can leave it out.

Integer variables

The variables to which we were referring before are capable of representing any number, including numbers which contain a decimal point. There is another type of variable, however, which can only represent whole

numbers or integers, which are numbers with no decimal points. This type of variable is called an integer variable.

Like normal variables, integer variables can consist of virtually any combination of letters and numbers, and all the normal rules apply. Integer variables may be recognised by the per cent (%) sign which goes after them. For instance FRED% is an integer variable, whereas FRED is a normal, or numeric, variable.

A numeric variable and an integer variable can have the same name and can both contain different numbers (although the integer variable can only contain an integer, of course).

Try this short program which illustrates the differences between the two sorts of variables (remember to type NEW first)

```
10 SCNCLR
20 H = 56.276:H$I% = 56.276
30 PRINT H,H%
```

This program clears the screen, then assigns the value 56.276 to the variable H. It then gives H% the same value as H, but as H% is an integer variable it can only hold the number 56, as can be seen when line 30 displays the contents of both variables.

You may have noticed that a comma is used to separate the two variables in line 30. This comma causes the value of the variable H% to be displayed starting at the eleventh space across the screen.

String variables

Another type of variable is the string variable. This kind of variable can represent letters and other characters, as well as numbers, and is distinguishable from the other types of variables by the dollar symbol ($) which goes after it.

Try typing NEW and entering this program which uses string variables

```
10 SCNCLR
20 A$ = "HELLO":B$ = "EVERYBODY"
30 PRINTA$;" ";B$
```

As you can see from the program, the characters which a string variable is to represent must be enclosed in quotation marks.

There are certain things which you cannot do with variables. Here are some examples.

A = "FRED"	(a numeric variable cannot represent characters)
PRINTER = 3	(a variable cannot incorporate a command. . .)
RUNNER$ = "HELLO"	(and neither can a string variable, or an integer variable)
B% = A$	(a numeric or integer variable cannot represent the contents of a string variable, or vice-versa)

Summary

There are three types of variables. These are numeric variables, integer variables and string variables.

Numeric variables are represented by a letter or a group of letters and numbers (eg Z, FD2, H34) and may only contain numbers.

Integer variables are represented in the same way as mumeric variables but have a percent symbol after them (eg Z%, FD2%, H34%) and will only recognise integers (ie if a number with a decimal point is assigned to an integer variable, the digits after the decimal point will be ignored).

String variables are represented in the same way as numeric variables but have a dollar symbol after them (eg Z$, FD2$, H34$) and may contain any characters. The characters must be enclosed in quotation marks.

The LET command can be used to assign values to variables, but it is not necessary to use this command.

Now would be a good time to experiment a little with variables before we move on to the next chapter. It is important that you understand how to use them, so a little time spent making sure that you do understand them now could save hours of frustration later.

LIST

You have already used the LIST command to display programs on the screen, but LIST can do much more than you have so far used it for. It is possible to LIST only parts of a program, for instance if you wanted to see lines 30 to 90 inclusive then you would type

LIST 30-90

If you wanted to see all of the program up to line 70 then you would type

LIST -70

You may want to see only the last few lines of a program, line 100 and all those after it, for instance, in which case you would type

List 100-

In another case you may only want to see, for example, line 25, in which case you would simply need to type

LIST 25

It is possible to slow down the rate at which a program is LISTed, simply by holding down the Commodore key. If, at any time, you need to stop a program being LISTed you simply have to press the RUN/STOP key.

RUN

You have also used the RUN command before, and therefore know that it is used to start a program from the beginning. It is also possible to start a program at some other line other than the first. For instance, if you wanted to start RUNning a program from line 50 onwards then you would type

RUN 50

The RUN command also resets any variables back to zero, or empties them of any characters if they are string variables.

NEW

NEW is another command which you have already used. This command removes a program from memory, and at the same time resets all variables. You should be careful when using this command as once you NEW a program it is gone for good.

INPUT

The INPUT command is very useful in programs as it allows the computer to ask questions of a person and then act on the answers. What the INPUT command actually does is waits for the user to type in an answer, and then stores that answer in a variable. Here is a short program which illustrates the INPUT command

```
10 SCNCLR
20 PRINT"HELLO, WHAT IS YOUR NAME";:INPUT NAME$
30 PRINT"TYPE IN ANY NUMBER";:INPUT A
40 SCNCLR
50 PRINT"HELLO";NAME$
60 PRINT"YOU TYPED IN THE NUMBER";A
```

When you RUN this program the screen will clear and you will be asked to type in your name (notice the question mark which appears — the computer displays this automatically when it is waiting for you to enter something in an INPUT command). Nothing will happen until you type in your name and prese the RETURN key. As soon as you have done this, however, you will be asked to type in a number, and nothing will happen until you do (and press RETURN, of course). Once you have done this the screen will clear again, and the computer will say hello to you and tell you what number you entered.

If you look at the above program you may be able to see that your name is being stored in the variable NAME$, and the number which you type in is stored in the variable A. This is because you are telling the computer to INPUT something from the keyboard and assign whatever is typed in to a variable. In this case we are using the variable NAME$ to represent your name, and the variable A to represent the number that you type in.

As you can no doubt see from the above program we were using a PRINT statement to ask for information to be typed in. We can however, incorporate a short message into the INPUT command. For instance, instead of having two commands such as

```
10 PRINT"WHAT IS 27 + 49";:INPUT Z
```

we could have

```
10 INPUT"WHAT IS 27 + 49";Z
```

The second line (with the message as part of the INPUT command) will display the message WHAT IS 27 + 49 on the screen and then wait for an answer. The reply is then assigned to the variable Z.

Summary

The INPUT command waits for something to be entered into the keyboard, and assigns whatever is typed in to a variable. A message can be incorporated into the INPUT command to save using a separate PRINT statement.

CHAPTER 4

Structuring your Programs

IF... THEN... ELSE

One of the most useful tasks which a computer can perform is the comparing of one value with another. This is essential in many programs, and is a feature which allows the computer to perform many useful tasks such as searching for a person's address and telephone number, which it does by comparing each name in a list with the name it is looking for.

There are three commands which combine to allow us to tell the computer to compare one value against another, and these are IF, THEN and ELSE. These three commands are known as a structure when they are used together like this. Here is an example of how they are used

```
IF Z = 42 THEN K = 9: ELSE K = 3
```

This tells the computer to see IF the variable Z represents the number 42, and if it does THEN assign the value 9 to the variable K, otherwise (ELSE) assign the value 3 to the variable K. If the value of Z is 42 then the ELSE part of the line is ignored,and the computer carries on to the next statement or line.

There are certain symbols which can be used to compare one value against another. These are '>' (greater than), '<' (less than) and '=' (equals). These symbols can be combined so that > = means greater than or equal to, and so on. Here are a few examples

```
IF FRED = 32 THEN JOHN = 30: ELSE JOHN = 22
```

IF the value of the variable FRED is 32 then assign the number 30 to the variable JOHN, ELSE assign the value 22 to the variable JOHN.

```
IF A$< > "Y" THEN SCNCLR
```

IF the string variable A$ does not represent the letter Y THEN clear the screen. If A$ does represent Y then the next line is carried out.

```
IF KK3> 99 THEN RUN
```

IF the value of the variable KK3 is greater than 99 THEN RUN the program.

There are two other operators. These are AND and OR. These operators are used in this way

IF A = 1 AND B = 1 THEN C = 1

IF the value of the variable 'A' is 1, AND the value of the variable B is 1 THEN assign the number 1 to the variable C.

IF A = 1 OR B = 1 THEN C = 1

If the value of the variable 'A' is 1, OR the value of the variable B is 1 (or both are 1) THEN assign the number 1 to the variable C.

Any commands following an IF statement will only be carried out if the conditions specfied are fulfilled.

Here is an example program which incorporates most of the commands which we have learnt so far

```
10 SCNCLR
20 INPUT "HELLO, HAVE WE BEEN INTRODUCED";IN$
30 IF IN$ = "Y" OR IN$ = "YES" THEN PRINT"I'M SORRY BUT I
CAN'T RECALL MEETING YOU"
40 INPUT"WHAT IS YOUR NAME";NAME$
50 PRINT"YOU CAN CALL ME COMMODORE!"
60 INPUT"WOULD YOU MIND IF I ASKED HOW OLD YOU
ARE";QU$
70 IF QU$ = "Y" OR QU$ = "YES" THEN PRINT"IN THAT CASE I
WON'T."
80 IF QU$ = "N" OR QU$ = "NO" THEN INPUT"PLEASE TELL ME
YOUR AGE";AGE$
90 PRINT
100 PRINT"WELL, I'M AFRAID I'LL HAVE TO GO NOW."
110 PRINT"I HOPE WE'LL MEET AGAIN, ";NAME$
```

And here is a line-by-line explanation of how the program works

Line 10: Clear the screen

Line 20: Display the message "HELLO, HAVE WE BEEN INTRODUCED" then wait for a response which is to be assigned to the variable IN$

Line 30: If the character assigned to the variable IN$ is Y or the characters assigned to the variable IN$ are YES then display the message "I'M SORRY BUT I CAN'T RECALL MEETING YOU"

Line 40: Display the message "WHAT IS YOUR NAME" then wait for a response and which is to be assigned to the variable NAME$
Line 50: Display the message "YOU CAN CALL ME COMMODORE!"
Line 60: Display the message "WOULD YOU MIND IF I ASKED HOW OLD YOU ARE" then wait for a response which is to be assigned to the variable QU$
Line 70: If the character assigned to the variable QU$ is Y or the characters assigned to the variable QU$ are YES then display the message "IN THAT CASE I WON'T."
Line 80: If the character assigned to the variable QU$ is N or the characters assigned to the variable QU$ are NO the display the message "PLEASE TELL ME YOUR AGE" then wait for a response which is to be assigned to the variable AGE$
Line 90: Display a blank line
Line 100: Display the message "WELL, I'M AFRAID I'LL HAVE TO GO NOW."
Line 110: Display the message "I HOPE WE'LL MEET AGAIN, ";NAME$

Summary

The IF...THEN...ELSE structure is used to compare one value against another and can compare all types of variables as well as fixed numbers or characters (e.g. IF A$ = "Y", IF ZZ = 22).

Special operators are used with IF...THEN...ELSE in order to compare things. These are '>' (greater than), '<' (less than) and '=' (equal to). These can be combined, e.g. > = means greater than or equal to.

AND and OR can be used so that more than one condition must be fulfilled before the computer can carry out a further ation — IF A = 2 AND B = 1 THEN PRINT"HELLO" — the computer will only display HELLO if the variable A represents the number 2 AND the variable B represents the number 1.

FOR...NEXT loops

It is often necessary to carry out the same instructions several times. For instance, you may use a program such as this to display the multiples of 15 up to 10*15

```
10 SCNCLR
20 PRINT 1*15
30 PRINT 2*15
40 PRINT 3*15
```

```
50 PRINT 4*15
60 PRINT 5*15
70 PRINT 6*15
80 PRINT 7*15
90 PRINT 8*15
100 PRINT 9*15
110 PRINT 10*15
```

Typing in a program such as this, although it works perfectly well, is tedious, and it uses up a lot of the computer's memory. To help you when you want to carry out a series of instructions several times, there are two commands called FOR and NEXT. These commands work together like this

```
10 FON N = 1 TO 10
20 PRINT N*15
30 NEXT N
```

This very short program consisting of only three commands replaces the first program with its ten PRINT statements. This is how the program works

Line 10: Tells the computer to start repeating all the instructions between the FOR and NEXT command ten times.
Line 20: Multiplies the current value represented by the variable N by 15 and display the result on the screen.
Line 30: If the value represented by the variable N is less than ten then the value of N is increased by one and the program goes back to line 20.

If you now try changing line 10 to

```
10 FOR N = 20 TO 30
```

and RUN the program then you will see the multiples of fifteen from 20*15 up to 30*15. This is because we have told the computer that we want the value of the variable N to start at 20 and be increased by one each time it loops round until the value of N reaches 30.

Another command, STEP, may be added to the end of the FOR command. This command tells the computer how much you want added to the variable involved in the loop each time. For example, if you change line 10 to

```
10 FOR N = 20 TO 30 STEP 2
```

and RUN the program then you will see the even multiples of fifteen from 20 to 30.

You can even go backwards through a loop by using a negative STEP. If you change line 10 to

```
10 FOR N = 30 TO 20 STEP -1
```

and RUN the program you will see the multiples of fifteen appear on the screen, but this time they start at 30*15 and work backwards to 20*15.

FOR. . . NEXT loops can be nested — put one inside the other. Try this example program

```
10 FOR N = 1 TO 10
20 FOR M = 1 TO 500:NEXT M
30 PRINT N
40 NEXT N
```

When you RUN this program you will see the numbers one to ten being displayed very slowly. The reason for this is we have a nested loop on line 20 which just goes round and round 500 times without doing anything. This slows down the program so that the outer loop carries out its function slowly. Here's another example of nested loops

```
10 SCNCLR
20 FOR N = 1 TO 12
30 PRINT"THE";N;"TIMES TABLE":PRINT
40 FOR M = 1 TO 12
50 IF M< 10 THEN PRINT" ";
60 PRINT M;"X";N;" = ";M*N
70 NEXT M
80 FOR Z = 1 TO 2000:NEXT Z
90 SCNCLR
100 NEXT N
110 RUN
```

This program contains two nested loops, one from lines 40 to 70, and the other on line 80. The main loop, from lines 20 to 100 chooses which table is being displayed. This starts at one and increases each time by one until the variable N reaches twelve.

The first nested loop uses the variable M and it also ranges from one to twelve. Line 50 displays a space if the value of M is less than ten,which makes the display look better — you can see exactly what it does by taking out this line and RUNning the program. The value of M is multiplied by the valueof N each time, so that all the multiples of N up to twelve times the value of N are displayed.

The second nested loop, on line 80, uses the variable Z and is a delay

loop. This loop just counts up to two thousand to give you enough time to read what is on the screen.

The screen is cleared for each times table by the SCNCLR command on line 90. Once the program has finished it restarts at the RUN command on line 110.

Summary

FOR...NEXT loops are used to carry out a set of instructions several times. The value of the variable used in the loop is increased by one each time it reaches the NEXT statement.

The STEP command can be added to the FOR command to alter the amount by which the value of the variable is increased eg. STEP 3 would add three to the variable each time,and STEP −2 would subtract two from the variable.

The NEXT command always marks the end of the loop. The name of the variable may be added (e.g. NEXT N), but this is not essential.

GOTO

As you should now know, when the computer carries out a program it works through it in the numerical order of the line numbers. It is possible to change this order, however, so that the computer can be made, for instance, to carry out line 10, then carry out lines 100 to 150, then return to line 20 and carry on from there.

The GOTO command tells the computer to GOTO a line number and carry on with the program from there. Try this short example program.

```
10 PRINT" #";
20 GOTO 10
```

If you RUN this program then you will see hash symbols '#' being PRINTed on the screen one after the other, non-stop. To stop this unending program you must press the RUN/STOP key.

The reason the program goes on and on is that after the hash symbol has been PRINTed by line 10, line 20 tells the computer to GOTO line 10, so another hash is PRINTed, and the computer goes back to line 10, and so on.

GOSUB and RETURN

The GOSUB command is very similar to GOTO. It tells the computer to GO to a SUBroutine and carry on with the program from there. If the

computer then reaches a RETURN command then it will carry on with the program from the next command after the GOSUB command.

A subroutine is a program within a program, in other words it's a short piece of program which carries out a specific task and can be used again and again. Being able to GOSUB to these routines saves a lot of typing as they need to be typed in only once.

Here is a short program which illustrates GOSUB and RETURN

```
10 PRINT"ROUTINE ONE"
20 PRINT"**********"
30 GOSUB 50
40 GOTO 10
50 PRINT"ROUTINE TWO"
60 PRINT"# # # # # # # # # # #"
70 RETURN
```

When you RUN this program you will see the following appear on the screen

```
ROUTINE ONE
**********
ROUTINE TWO
# # # # # # # # # # #
ROUTINE ONE
**********
ROUTINE TWO
# # # # # # # # # # #
```

except that this will be PRINTed over and over again until you stop the program.

The program works in this way

Line 10: Display the message 'ROUTINE ONE'
Line 20: Display the message '**********'
Line 30: Go to the subroutine starting at line 50
Line 40: Go to line 10 and carry on with the program from there
Line 50: Display the message 'ROUTIINE TWO'
Line 60: Display the message '# # # # # # # # # # #'
Line 70: Return to the command immediately following the GOSUB command which called this routine (in this case the computer will go to line 40 as this contains the first command following the GOSUB command)

You may already have realised that if a line such as this occurs in a program

```
100 GOTO 50:PRINT"HELLO"
```

then the PRINT statement will never be carried out, because as soon as the computer reaches the GOTO command it will go to line 50 and carry on with the program from there. However, if line 100 read

```
100 GOSUB 50:PRINT"HELLO"
```

then the PRINT statement will be carried out as soon as the computer is told to RETURN from the subroutine starting at line 50, as the PRINT statement is the next command after the GOSUB command.

One thing that you cannot do with a GOTO or GOSUB command is to replace the line number with a variable so that if you want to GOTO 10 you cannot replace the 10 with a variable, even if that variable has the value ten.

The GOTO and GOSUB can, of course, be used with the IF... THEN...ELSE structure, which can be very useful.

Summary

The GOTO command tells the computer to GOTO a certain line number and carry on with the program from there.

The GOSUB command tells the computer to GO to a SUBroutine starting at a certain line number and carry on with the program from there. When a RETURN command is reached the computer will return to the statement immediately after the last GOSUB statement.

Any commands after a GOTO command on the same line will not be carried out.

The line number after the command cannot be replaced with a variable.

Editing your programs

Until now whenever you have made a mistake in your programs you have had to type the line again. It is possible, however, to alter your programs without doing this. Type in this program exactly as it is printed

```
10 SCMCLR
20 PRIINT"HELLOO"
30 GTO20
```

Obviously, this program is full of mistakes, but it would not be very convenient to have to type it all in again. Fortunately we do not have to do this, we can edit the program instead.

Move the cursor to the M in SCMCLR and then type N and then RETURN. Line 10 will then have been corrected.

The next mistake is in line 20. This line has two mistakes. To correct it you should first move the cursor over the N of PRIINT, and then press the INST/DEL key. Then you should move the cursor over the second O, and then press the INST/DEL key again. Press RETURN and line 20 will be corrected.

The GOTO command in line 30 has an O missing. To insert this letter you should move the cursor to the T and then press SHIFT and INST-/DEL. A space will then be opened up ready for you to type an O in the right place. You should then press RETURN, type LIST and your program will be displayed in its corrected form.

If you decide half-way through editing a line that you want it left as it was then you can cancel the alterations you have made either by pressing SHIFT CLEAR/HOME (which clears the screen), or pressing SHIFT and RETURN. You could also just move the cursor off the line which you are correcting, because no alterations are permanent until the RETURN key is pressed.

Storing your programs on tape

If you have wanted to re-use one of the programs which you have entered earlier on in this book then you will probably have found it a nuisance to have to keep typing it in each time you want. To avoid this you can store programs on normal cassette tape.

You will probably already have a Commodore 1531 Datassete. If you do not then it would be a good idea to buy one, even if you want to use disk drives, as a number of the programs which are available are on cassette only. The Datassette should be plugged into the socket marked CASS-ETTE at the rear of the computer (you should switch off the computer while doing this). All you need now is some blank tapes — most makes of audio cassettes work perfectly as long as they are of normal bias, but you can buy special computer cassettes.

Once you have everything connected up and a suitable tape in the Datas-sette you should fast-forward the tape until the leader is past the read head of the Datassette. You are now ready to save a program. This program will do for a test

```
10 SCNCLR
20 PRINT"THIS PROGRAM HAS BEEN SAVED ON TAPE AND"
30 PRINT"SUCCESSFULLY LOADED BACK"
40 FOR N = 1 TO 25
50 PRINT"****************************************"
60 NEXT N
70 FOR N = 1 TO 2000:NEXT N
80 RUN
```

Now all you have to do is type

SAVE"PROGRAM"

You will then be asked to PRESS PLAY AND RECORD ON TAPE, which you should do in the same way as when you are recording on a normal tape recorder. The screen will then go blank and the tape motor will start up. After a short while the screen will return to normal and the tape will stop — your program has been recorded on tape.

The command SAVE"PROGRAM" tells the computer to make a copy of the program currently in memory onto tape and give that program the name PROGRAM (giving a program a name makes it easier to find later). As you can see the name must be enclosed in quotation marks.

VERIFYing the program

You can make sure that the program has been recorded correctly by using the VERIFY command. Rewind the tape and type

VERIFY"PROGRAM"

The computer will tell you to PRESS PLAY ON TAPE, and as soon as you have done this the screen will go blank and the tape motor will start. The computer will then search for a program which has been saved on tape under the title PROGRAM. When it finds the program the computer will display the message

FOUND PROGRAM

VERIFYING

on the screen and will then pause. You can save time at this point by pressing the Commodore key which tells the computer not to wait. The computer will then procede to compare the program which is on the tape with the one which it currently has in memory. If they are the same (which they should be because you still have the program which you saved in memory) then the screen will eventually return to normal and the computer will inform you that everything is OK. If the program was not saved correctly then it will display the message

VERIFY ERROR

If this happens you should try rewinding the tape, wiping it clean and starting again. If you continue to be unable to save the program then try another

tape. If, after several different tapes, you are still unable to successfully save a program then you should take the computer and Datassette to your local Commodore stockist and have them checked.

LOADing the program

You will now, I hope, have your program saved on tape. Type NEW and then type

```
LOAD"PROGRAM"
```

The computer will ask you to PRESS PLAY ON TAPE, as it did when you were VERIFYing, and as soon as you do this the screen will go blank and the computer will start looking for the program called PROGRAM. As soon as it finds the program the computer will display the message

```
FOUND PROGRAM

LOADING
```

Again there will be a pause, which can be cut short by pressing the Commodore key. The screen will then go blank while the computer loads the program. Once the OK message appears you can type LIST or RUN and your program will be there.

The name following the LOAD, SAVE and VERIFY commands is not compulsory, although it is a good idea to give each program a name. For instance, if you just type LOAD and press the RETURN, the computer will load the first program it comes across into its memory.

You may add two extra numbers to the end of the LOAD and SAVE commands. These numbers are the device number and the relocate flag. For instance, the command

```
SAVE"GRAPH",8,1
```

would save the program in memory onto disk, which has device number 8, and give it the name GRAPH. For an explanation of the different device numbers, please refer to your manual. Adding ',1' tells the computer to add an End Of Tape marker to the end of the program. This means that if the computer is searching for a program on tape, and it comes across an End Of Tape marker (EOT) it thinks that it has come to the end of the tape, and a FILE NOT FOUND ERROR is given.

Similarly the command

```
LOAD"GRAPH",1,1
```

would cause the computer to load in the program with the name GRAPH from tape (the first ',1' tells the computer to load from tape), and to load it back to the memory location from which it was saved. Adding ',1' to the end of the LOAD command makes the computer load the program back into the same area of memory as the one from which it was saved. If this is left off the computer will load the program back at the start of BASIC memory. You will only need to use this command to load machine code programs which will be explained later, as these are sometimes stored in different areas of memory.

Summary

The SAVE command is used to make a copy of the program currently in memory onto tape. The program can be given a name to make it easier to locate on the tape later.

The program name can be made up of any characters and may be up to sixteen characters long. The program name must be enclosed in quotation marks.

The VERIFY command is used to check that the program saved on tape is exactly the same as the one currently in memory. This can be used to find the end of a program simply by typing VERIFY"program name" and wait until the computer gives you a verify error.

The LOAD command is used to read a program back from tape and in to the computer's memory. The program name can be used, but is not essential if you know exactly where the program is on the tape.

Some machine code programs need to be loaded as LOAD"program name",1,1.

CHAPTER 5

More Ideas

INT

The INT command is used to round down numbers. For instance, if you typed

```
PRINT INT(32.3)
```

then you will see the number 32 appear on the screen, because the number 32.3, when rounded down, is 32. INT will always round in a number down, so 98.1 will become 98, −54.87 will become −55 and so on.

RND

The RND command, like INT, is a 'function', which is a command which takes a number and does something with it, before giving you back the result.

The RND function chooses a random number between 0 and 1. The way in which this number is chosen depends on the seed which we choose. This seed is enclosed in brackets following the RND function, and may either be a negative number, a positive number, or zero.

If we use a positive seed, such as

```
PRINT RND(1)
```

then the computer will choose a random number, or at least as random as is possible (the numbers will repeat themselves after several thousand tries).

A negative seed, like this

```
PRINT RND(-1)
```

re-seeds the random number generator. When a random number is chosen, it is selected from a long series of numbers. Each time a random number is needed it is taken from the next number in that series. Using a negative seed alters the position in that list that the number is taken from. In the above example you should receive the answer 2.99196472E−08, but if you used

another negative seed, eg −2, you would get a different number.

Finally, if you use a zero seed, like this

```
PRINT RND(0)
```

then the random number is chosen according to the state of the built-in clock. This method of choosing a random number is probably the best for most purposes as it is virtually impossible for the sequence to be repeated.

This program choses thirty random numbers between 1 and 10, ten chosen from each way of choosing a random number

```
 10 SCNCLR
 20 PRINT"POSITIVE SEED":X = 1
 30 GOSUB 80
 40 PRINT"NEGATIVE SEED":X = −1
 50 GOSUB 80
 60 PRINT"ZERO SEED":X = 0
 70 GOSUB 80:END
 80 FOR N = 1 TO 10
 90 PRINT INT(RND(X)*9) + 1;
100 NEXT N
110 FOR N = 1 TO 4000:NEXT N
120 SCNCLR:RETURN
```

You may notice that the random number chosen in line 90 is multiplied by nine and then rounded up. This has the effect of choosing a whole number between zero and nine. Adding one to the result has the effect of choosing a number between one and ten.

CHAR

The CHAR command is very similar to the PRINT command in that it allows us to display characters on the screen. CHAR differs from PRINT, however, in that it allows us to choose exactly where on the screen we want the characters to be displayed.

You may already have noticed that you can have 40 characters across and 25 lines down of text on the screen. This means that you can have up to 1000 characters on the screen at once.

Now, imagine that a grid has been drawn on your television screen, and that there are 40 squares across and 25 down. Above the top line of squares are written the numbers 0 to 39, and to the left of the left hand column of squares are written the numbers zero to 24, just like on a map. It is now possible to give any square on the screen a grid reference, so you can therefore locate any character on the screen and say exactly where it is. For

example, if there is a letter in the top left hand corner of the screen it is at 0,0 (zero squares across and zero squares down). If there is a letter in the bottom right hand corner of the screen then it is at 39,24 (39 squares across and 24 down).

Now that you know how the screen is divided up and how each character square is referred to, you can start to use the CHAR command. All you have to do is tell the computer which character square you want the first character of your message to appear, and then tell it what you want it to display. Try this example program

```
10 SCNCLR:INPUT"X CO-ORDINATE";X
20 INPUT"Y CO-ORDINDATE";Y
30 CHAR1,X,Y,"HELLO"
40 FOR N = 1 TO 4000:NEXT N
50 RUN
```

When you RUN this program you will be asked for an X coordinate, in other words the number of character squares across you want the message to start at. You will then be asked for the Y coordinate, or how many squares down you want the message to start at. You will probably find this difficult to understand at first, so try experimenting with differing X and Y coorindates to see what results you get. Once you have done this you will have the word HELLO appear on the screen, starting at the square you have just chosen.

If you look at line 30 of the program then you will see the CHAR command. You will see where to put the X and Y coordinates and the message, but there is also a number one straight after the CHAR command. This number tells the computer that you want to display the character in the current text colour. Changing this number to a 0 makes the character appear in the background colour, so you won't be able to see it.

As you can see, the 1, the X and Y coordinates and the message are all separated from each other by commas. The message can be of any length as long as the length of the whole command, including the message, is not more than eighty characters.

If you try changing line 30 to this

```
30 CHAR1,X,Y,"HELLO",1
```

and RUN the program then you will see the word HELLO appear in reverse characters. This is achieved by adding ',1' after the message. If nothing is added, or ',0' is added, then the characters appear as normal.

Summary

The screen is split up into 1000 character squares — 40 across and 25 down.

This allows us to give each square in the grid a coordinate.

The CHAR command is used to allow us to choose where we want to display any characters or messages.

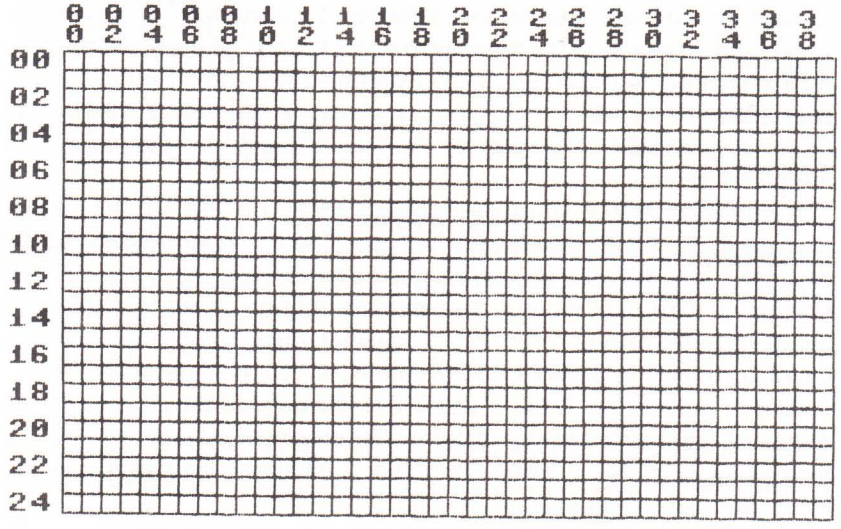
The message is enclosed in quotation marks, just like with the PRINT statement.

Adding ',1' after the message makes the message appear in reverse characters.

Characters may be displayed in the current text colour (CHAR 1,), or the background colour (CHAR 0).

CHAR Grid

Here is a CHAR grid showing the coordinates of each character square on the screen. For instance, the character square 10 squares across and 4 down has the coordinates 9,3.



Using joysticks

Joysticks are versatile pieces of equipment, and can be used for all kinds of purposes, although the most obvious, and most common, is for playing games. You may connect up to two joysticks to your computer. These should be plugged into the sockets marked JOY 0 and JOY 1.

Reading the joysticks is fairly simple — you simply use the JOY function, which will give you these results, depending on the position of the joystick.

UP
1

8 2

LEFT 7 0 3 RIGHT

6 4

5

DOWN

So, if the joystick which you are reading is pointing up then the JOY function will return the value 1, if the joystick is pointing downwards to the right then the value 4 will be returned. If the fire button is being pressed then 128 will be added to the direction number.

Now that you know what results you will get when the joysticks are read, it would be useful if you knew how to read them. Try typing in this program

```
10 SCNCLR
20 PRINT"RIGHT JOYSTICK LEFT JOYSTICK"
30 RI = JOY(1):LE = JOY(2)
40 CHAR 1,5,3,RI:CHAR 1,25,3,LE
50 GOTO 30
```

The right joystick is read by the function JOY(1), while the left joystick is read by the function JOY(2) (this is a little confusing as the joystick sockets are marked JOY 0 and JOY 1).

This program reads the value of the right joystick into the variable RI and the value of the left joystick into the variable LE. These values are then displayed on the screen. If the fire button is being pressed on either joystick then the direction number is increased by 128, as you will see if you try pressing the fire button. When the joystick is centred the value displayed will be zero.

GET

You already know one way of obtaining information via the keyboard, and that is to use the INPUT command. However, can you imagine what it would be like trying to play a game and having to press RETURN after each move? This is what we would have to do if we had to use the INPUT command.

Luckily for us, however, we have two commands which can be used to

test if a single key is being pressed. These are GET and GETKEY. Try this short program

```
100 SCNCLR
110 PRINT"DO YOU WNAT ANOTHER GO (Y/N)?";
120 GET A$:IFA$ = ""THEN GOTO 120
130 IF A$ = "N" THEN END
140 IF A$ = "Y" THEN RUN
150 GOTO 120
```

When you RUN this program you will be asked DO YOU WANT ANOTHER GO (Y/N)? and then nothing will happen unless you press the Y or N key. If you press the N key then the program will stop, but if you press Y then the program will RUN again. Pressing any other key has no effect.

Line 120 contains the GET command. As you can see, a variable must be used with GET (in this case A$), because what the GET command does is scan the keyboard to see if a key is being pressed, and if one is then the character on that key is assigned to the variable. If none of the keys is being pressed then the variable will remain empty. Line 120 tests to see if A$ is empty, and if it is then line 120 is carried out again and again until a key is pressed.

Lines 130 and 140 test to see which key is being pressed. If neither the Y nor the N key is pressed then nothing happens and line 150 tells the computer to go back to line 120 to have another go.

The GET command does not cause the program to stop, as the INPUT command does, and can only read one character at a time. This means that the program will carry on whether a key is pressed or not.

GETKEY

The GETKEY command is similar to GET, but in this case it waits for a key to be pressed, and the program will not continue until it has. This saves us having to test to see if there is anything in the string variable that we use, but there is the disadvantage that the program stops. Using GETKEY we can change line 120 to:—

```
120 GETKEY A$
```

The program still does the same job, but we no longer have to test to see if A$ represents anything, because GETKEY will wait until a key is pressed before assigning the character on that key to the variable A$.

Summary

The GET command will scan the keyboard and assign the character on any key that is being pressed to a string variable. If no key is being pressed then that variable will be empty.

The GET command does not stop the program, unlike the INPUT command, and does not need the RETURN key to be pressed.

The GETKEY command is similar to the GET command but stops the program until a key is pressed.

DO...LOOPs

You have already come across one form of loop, the FOR...NEXT loop, but there is one other type of loop — the DO...LOOP. This form of loop is, in some ways, simpler to understand than the FOR...NEXT loop, but is used less often. Here is an example of a DO...LOOP

```
10 DO
20 PRINT"HELLO"
30 LOOP
```

If you RUN this program the computer will repeatedly display the word HELLO until you stop it by pressing the RUN/STOP key. This is because we are telling the computer to display the word HELLO then to LOOP back round to the command immediately after the DO statement (which happens to be the PRINT statement again) and start again.

This may seem to be of no use at all. However, if you make these alterations to the program you may think otherwise.

```
10 DO UNTIL A = 10
15 A = A + 1
```

When you RUN the program this time you will see the computer display the word HELLO ten times and then stop. This is because we are now telling the computer to carry out everything between the DO and LOOP statements UNTIL the value of the variable 'A' reaches 10. Since the value of A starts at zero and is increased by one each time the computer LOOPs round, the word HELLO is only displayed ten times. If you altered line 10 to

```
10 DO UNTIL A = 20
```

then the word HELLO would be displayed 20 times.

Another form of the DO...LOOP is the DO WHILE...LOOP. This is

very similar to the DO UNTIL . . . LOOP, as you will see if you alter line 10 to

```
10 DO WHILE A< 10
```

When you RUN the program you will see the word HELLO displayed ten times, exactly the same as before. This time the computer will carry out everything between the DO and LOOP statements only WHILE the value of the variable A is less than 10. As soon as the value of A reaches 10 the computer will leave the loop.

The WHILE and UNTIL statements can be put after the DO statement or after the LOOP statement, so you could change lines 10 and 30 to

```
10 DO
30 LOOP WHILE A< 10
```

and the program will work in exactly the same way.

It is sometimes necessary to leave a DO . . . LOOP half way through. To do this we use the EXIT statement. Try adding this lines to your program.

```
25 GET A$:IF A$ = "A" THEN EXIT
```

You can now EXIT from the program by pressing the A key. This is because the computer is being told to scan the keyboard, and if the A key is being pressed then it should EXIT from the loop.

It is possible to nest DO . . . LOOPs, in the same way as you nest FOR- . . . NEXT loops. Here is an example of nested DO . . . LOOPs

```
10 SCNCLR
20 DO UNTIL Y = 25
30 DO UNTIL X = 40
40 CHAR 1,X,Y,"*"
50 FOR N = 1 TO 50: NEXT N
60 CHAR 1,X,Y," "
70 X = X + 1:LOOP
80 X = 0:Y = Y + 1:LOOP
```

The program works like this

Line 10: Clear the screen
Line 20: Carry out everything between the DO and LOOP statements until the value of the variable Y reaches 25
Line 30: Carry out everything between the DO and LOOP statements until the value of the variable X reaches 40

Line 40: Display a star in Xth character square across and the Yth character square down
Line 50: Delay loop
Line 60: Display a space in the Xth character square across and the Yth character square down
Line 70: Add one to the value of the variable X then LOOP round to the statement after the last DO statement
Line 80: Assign the value zero to the variable X, add one to the value of the variable Y and loop back round to the first DO statement

The inner loop is carried out before the outermost loop, and is carried out 40 times for each increment of the outer loop. The inner loop controls the horizontal position of the star, so carrying this loop out 40 times makes the star move right across the screen one character step at a time. The star is moved down one line for each time the inner loop has completed 40 circuits until it reaches the bottom of the screen.

Summary

All statements between the DO statement and the LOOP statement are carried out UNTIL a condition is fulfilled or WHILE a condition is fulfilled.

The UNTIL or WHILE statements can go after the DO statement or after the LOOP statement.

The EXIT statement will make the computer stop carrying out the loop, whether it has finished or not.

DO...LOOPs can be nested in exactly the same way as FOR...NEXT loops.

DIM and array variables

The array variable is a slightly more complex form of variable than those that you have been using up to now. Array variables are more useful in that they have a number which is similar to an index number.

Array varaibles are difficult to understand at first, but a good way to think of them is this. Imagine a book with 12 pages, numbered 0 to 11. On each page is written one number. This book represents a numeric array variable. If for instance we wanted to see what was written on page 5 of the book you would take the book, turn to page 5 and see what was there.

An array variable is similar to the book with its numbered pages. They work in a similar way in that if, for instance you had an array called BOOK, and you wanted to see what the fifth number represented by BOOK was you would type

```
PRINT BOOK(5)
```

The computer would then look for the array variable BOOK and see what the fifth number represented by that array is, and then display that number on the screen.

Unfortunately, arrays take up a lot of the computer's memory, so it is important that we tell the computer how many arrays we want to use, and how much we want to store in those arrays. To do this we must use the DIMension command. For instance, if you want to store 14 numbers in an array called BOOK then you would use a line like this

```
10 DIM BOOK(13)
```

This tells the computer to reserve enough memory for 14 numbers (0 to 13 is fourteen numbers). We can now add to the program so that it asks for fourteen numbers and then PRINTs them out

```
20 SCNCLR
30 FOR N=0 TO 13
40 INPUT"PLEASE TYPE IN A NUMBER";BOOK(N)
50 NEXT N
60 SCNCLR
70 FOR N=0 TO 13
80 PRINT BOOK(N)
90 NEXT N
```

When you RUN this program the computer will ask you to type in fourteen numbers and then the screen will clear and all the numbers that you typed in will be displayed on the screen. Here is a plain English version of the program.

Line 10: Reserve enough memory to store 14 numbers in the array variable book.
Line 20: Clear the screen
Line 30: Start repeating all commands between the FOR and NEXT commands fourteen times
Line 40: Display the message PLEASE TYPE IN A NUMBER, wait for a number to be entered and then assign that number to the Nth part of the array variable BOOK.
Line 50: Marks the end of the FOR . . . NEXT loop
Line 60: Clear the screen
Line 70: Start repeating all commands between the FOR and NEXT commands fourteen times

Line 80: Display the number represented by the Nth part of the array variable BOOK
Line 90: Marks the end of the FOR. . . NEXT loop

The above program used one-dimensional array variables, or, comparing them to the book, there was only one book. However, we can also use two-dimensional arrays. Comparing these to our book, this means that we can not only choose which page to look at, but we can also choose which book from a whole shelf of books. The array has two index numbers, so we can say which book and which page we want to refer to.

Two dimensional arrays are useful for tables. For instance, if we wanted to put the results of a series of races on a table it might look something like this.

	First	Second	Third
100m (Boys)	Paul	Mark	Ray
400m (Boys)	Kevin	Martin	William
100m (Girls)	Sarah	Jane	Carol
400m (Girls)	Maria	Claire	Julie

This kind of table can be represented by a two-dimensional array quite easily. To do this we would need a two-dimensional array which can hold twelve elements (each number or set of characters assigned to an array is called an element), three elements across and four elements down. The array would obviously need to be a string array because we need to store names. This program does the job quite well

```
10 SCNCLR
20 DIM PLACE$(2,3)
30 PLACE$(0,0) = "PAUL":PLACE$(1,0) = "MARK":PLACE$(2,0) =
"RAY"
40 PLACE$(0,1) = "KEVIN":PLACE$(1,1) = "MARTIN":PLACE$
(2,1) = "WILLIAM"
50 PLACE$(0,2) = "SARAH":PLACE$(1,2) = "JANE":PLACE$(2,2) =
"CAROL"
60 PLACE$(0,3) = "MARIA":PLACE$(1,3) = "CLAIRE":PLACE$
(2,3) = "JULIE"
70 PRINT TAB(9);"RACE RESULTS":PRINT
80 PRINT"1) 100M (BOYS)":PRINT"2) 400M (BOYS)":PRINT"3)
100M (GIRLS):PRINT"4) 400M (GIRLS)"
90 PRINT:INPUT"WHICH RACE DO YOU WANT THE RESULTS
FOR";RACE
```

```
100 IF RACE> 4 OR RACE< 1 THEN GOTO 90
110 INPUT"WHICH POSITION DO YOU WANT TO KNOW";PS
120 IF PS> 3 OR PS< 1 THEN GOTO 110
130 RACE = RACE – 1:PS = PS – 1
140 PRINT"THE PERSON WHO ACHIEVED THAT PLACE IN
THAT RACE WAS"
150 PRINT PLACE$(PS,RACE)
160 PRINT:PRINT"PRESS ANY KEY FOR ANOTHER RESULT"
170 GETKEY A$:SCNCLR:GOTO 70
```

The program works like this

Line 10: Clear the screen
Line 20: Reserve enough memory for a string array of the name PLACE$ with enough room for 3 elements by four elements
Line 30–60: Assign names to each element of the array variable PLACE$, eg assign the name PAUL to element (0,0) of the array PLACE$, assign the name MARK to element (1,0) of the array PLACE$
Line 70: Display the message RACE RESULTS on the screen with the first letter of that message in the 9th character square across then display a blank line
Line 80: Display the messages '1) 100M (BOYS)', '400M (BOYS)' etc
Line 90: Display a blank line then display the message 'WHICH RACE DO YOU WANT THE RESULTS FOR' and wait for a response which should be assigned to the variable RACE
Line 100: If the value of the variable RACE is bigger than four, or the value of the variable RACE is less than one then go to line 90 and carry on with the program from there
Line 110: Display the message 'WHICH POSITION DO YOU WANT TO KNOW' and assign the response to the variable PS
Line 120: If the value of the variable PS is bigger than three or the value of the variable PS is less than one then go to line 110 and continue with the program from there
Line 130: Subtract one from the value of the variable RACE and assign the result to the variable RACE then subtract one from the value of the variable PS and assign the result to the variable PS
Line 140: Display the message THE PERSON WHO ACHIEVED THAT PLACE IN THAT RACE WAS
Line 150: Find the value of element (PS,RACE) of PLACE$ and display it on the screen
Line 160: Display a blank line then display the message 'PRESS ANY KEY FOR ANOTHER RESULT'
Line 170: Wait for a key to be pressed then clear the screen and go to line 70 to continue with the program from there

The size of your arrays is restricted only by the amount of memory space you have available, so you could have an array such as A(4,4,4,4,4,4) if you had enough memory space. However, you are unlikely to need to use arrays much bigger than a three-dimensional array, e.g. A(5,5,5).

Summary

The DIM statement is used to reserve memory space for an array variable.

An element is a number or string of characters that has been assigned to one part of an array.

An array consists of a variable name with a series of index numbers following it. The index numbers should be enclosed in brackets and tell the computer which part of the array you are referring to.

The size of an array is limited only by the amount of memory you have available.

CHAPTER 6

Tidying Up

CHR$

You may have wondered exactly how the computer stores characters in its memory. Obviously, it cannot actually store the shapes of each character in a program in its memory, so it has to give each character a code. For instance, the letter A has the code 65, the heart symbol has the code 115 and so on.

The CHR$ command allows us to use these character codes in order to display characters. If you type

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

you will see a letter A appear on the screen, because the letter A has the code 65. Similarly, if you type

```
PRINT CHR$ (115)
```

you will see a heart appear on the screen. You could try experimenting with different character codes and see what effects they have (choose codes from the table in Appendix C, but do not use a number higher than 255 because there are no more characters after the one with the code 255).

Some CHR$ codes are control codes. These are mainly from 0 to 31, although there are some others. Control codes are special characters which move the cursor around, change the text colour and so on.

TAB

The TAB command is always used with the PRINT command, and is used to decide which column (or how many character squares across) you want to start PRINTing in. For instance

```
PRINT TAB(30);"HELLO"
```

will display the message HELLO, with the H appearing in the 30th character square across, the E in the 31st character square and so on.

One important thing to remember is that if, for example you typed in this command

PRINT TAB(30);"HELLO";TAB(10);"THERE"

where the second word is to be displayed *behind* the first word then the second word will be PRINTed on the next line down from the first, although it will start in the 10th character square across.

DELETE

Imagine that you are writing a long program and you decide that you do not need a certain routine that is about fifteen lines long. To delete them by typing the line number of each line and then pressing RETURN would take quite a while. To save our fingers from this extra typing your computer has been provided with a command which will delete several program lines at once. This command is the DELETE command.

The DELETE command is used in a very similar way to LIST, except instead of displaying the program lines, DELETE removes them from the program. For instance, if you want to DELETE lines 90 to 150 inclusive then you have to type

DELETE 90–150

To DELETE all the lines up to line 75 inclusive you would type

DELETE –75

And to DELETE all the lines from line 5010 onwards inclusive you need only type

DELETE 5010–

RENUMBER

The RENUMBER command is extremely useful for when you are writing your own program as it allows you to tidy up the line numbers. Imagine, for instance, that you are writing your own program, and have been numbering the lines in steps of ten all the way through. You then discover that you need to add 13 lines between lines 50 and 60. There is obviously no way that they will fit, so what do you do? You RENUMBER the lines in steps of twenty and you have plenty of room to add the extra lines, as well as any others you might need later.

You can RENUMBER a program in one of several ways. These are

RENUMBER: Renumbers the whole program, with the first line becoming line 10, and with the following lines in increments of 10.

RENUMBER 50,,7: Renumbers the program with line 7 becoming line 50 and all following lines in increments of 10.

RENUMBER 100,20,15: Renumbers the program with line 15 becoming line 100 and all the following lines in increments of 20.

RENUMBER 200,40: Renumbers the program with the first line number becoming 200 and the following lines in increments of 40.

REM

When writing a program of your own, especially one which is quite long and has several subroutines, it is often useful to be able to put comments here and there to help you remember what each part does. The REM (which is short for REMark) allows you to make such comments, which are ignored by the computer. For instance, this line would help you to remember that the routine following it is to move a bat left

530 REM MOVE BAT LEFT

You can put anything you like after a REM, but the computer will ignore that line.

END

We have already used the END command a couple of times in our programs, so we should know that it tells the computer to stop carrying out the program, even if it has not reached the last line. This may seem to be of no use, but you do need to stop a program before the last line if there are subroutines at the end of the program.

STOP

The STOP command is similar to END except that it is possible to re-start the program, and you are also told which line the program STOPped at.

CONT

The CONT command is used to CONTinue a program after it has been STOPped, or after you have pressed the RUN/STOP key. This command

will CONTinue with the program from where it left off and will not reset any variables. In some cases it is not possible to CONTinue a program — if you have altered a program line after STOPping the program for instance. In these cases you will be given a CAN'T CONTINUE ERROR.

Windows

Windows allow you to work in an area of the screen so that the rest of the screen remains undisturbed. Anything that normally takes place on the full-sized screen display will instead take place in the window. For example, clear the screen (by pressing SHIFTed CLEAR/HOME), then press the down arrow key five times. Press the ESC key, then press the T key. Press the down arrow key another eight times, then press the right arrow key ten times. Press ESC again and then B. Finally, press the HOME key the cursor will jump to the top left hand corner of the window.

You can type in programs, as well as LIST and RUN them, in a window, and anything you type in will appear in this window, as you will probably see. The window is effectively working as a smaller version of the screen. To get out of the window again you should press the HOME key twice.

Pressing ESC then T marks the top lefthand corner of the window, while pressing ESC and then B marks the bottom righthand corner of the window. You will notice that you cannot move the cursor up or to the left after marking the top lefthand corner of the window, and you cannot move the cursor out of the window in any direction once it has been defined, until you press the HOME key twice.

Windows can be set up quite easily from within a program. To do this you need to know character code for the ESC key which is character code 27. Once you know this you can set up a window wherever you want one, like this ([5X CRSR DOWN] means press the down arrow key five times, and [4X CRSR RIGHT] means press the right arrow key five times)

```
10 SCNCLR
20 PRINT"[5X CRSR DOWN][4X CRSR RIGHT]";CHR$(27);"T";
30 PRINT"[10X CRSR DOWN][14X CRSR RIGHT]";  CHR$(27);
"B[HOME]";
40 FOR N = 1 TO 100:PRINT"A WINDOW!";:NEXT N
```

As you can probably see, setting up a window from within a program is similar to setting up a window from command mode. The size and position of the window is still defined by moving the cursor to the top lefthand corner of the window, and then to the bottom righthand corner. The only difference that the cursor controls are enclosed in quotation marks, and whenever we would normally press the ESC key we put CHR$(27) instead,

and where we normally type T or B we enclose the same letter in quotation marks.

Windows are useful if you are converting a BASIC program from a different computer onto your computer. If the size of the text screen on the other computer is less than 40 characters across by 25 lines down, then you can set up a window of the same size as the screen on the other computer. All printing will then go inside the window, to save you having to work out too much.

The ESC Key

We have already used the ESC key in order to define the size of a window, but this versatile key has many other uses. This key, when used with one of the normal letter keys, is used to activate special functions on your computer. A list of these functions is shown below, followed by a brief description of what each one does

ESC A	Turn on insert mode
ESC B	Set the bottom righthand corner of the window to the current cursor position
ESC C	Turn off insert mode
ESC D	Delete the line which the cursor is on
ESC I	Insert a line
ESC J	Move the cursor to the start of the current line
ESC K	Move the cursor to the end of the current line
ESC L	Turn on screen scroll (normally on)
ESC M	Turn off screen scroll
ESC N	Return screen to normal size
ESC O	Turn off flash, reverse, insert and quote modes
ESC P	Erase everything on current line before the cursor
ESC Q	Erase everything on current line after the cursor
ESC R	Reduce the size of the screen display
ESC T	Set the top lefthand corner of the window to the current cursor position
ESC V	Scroll the screen up one line
ESC W	Scroll the screen down one line
ESC X	Cancel the ESC function

Pressing ESC then A (you should release the ESC key before pressing another) turns on the insert mode. This means that everything you type will be inserted into whatever is on the current line. For instance, if you type

```
10 PRINT"HELLO"
```

then move the cursor to the second set of quotation marks and type ESC then A. Whatever you now type in will be inserted before the quotation marks, which will move over to the right one space each time you type in another character. When you have finished inserting text into this line, press ESC then C and insert mode will be cancelled.

You may also delete a whole line and cause everything below that line to scroll up to fill the space. For instance, if you RUN the above one-line program for a while, then stop it, move the cursor to any position on the screen and type ESC then D you will see that the contents of the line which the cursor is on will be deleted, and everything on the screen below that line will scroll up.

The opposite of deleting a line is inserting a line. Pressing ESC the I will cause everything on the screen below the cursor to scroll down one line, leaving a blank line for you to type on.

It can sometimes be useful to move the cursor to the beginning of a line when it has half-way along one. The fastest, and easiest, way to do this is to type ESC then J. For instance, if you move the cursor to any point on the screen, preferably about half-way across, then press ESC and J, the cursor will jump to the beginning of the line that it is on.

You can also move the cursor to the end of the current line in a similar way. This is done by pressing ESC then K, in exactly the same way as when you move the cursor to the start of the line.

Something which can be very useful is the ability to turn off the screen scroll. If you type ESC then M, and move the cursor down to the bottom of the screen, you will see it re-appear at the top. This is known as wrap-around. If you LISTed a program now you would see that when the screen becomes full it does not scroll up to allow more to be displayed, instead the LISTing goes off the bottom of the screen and comes back at the top. Everything can be returned to normal by pressing ESC then L.

The ESC O function will cancel flash, reverse, quote and insert modes if they have been turned on. For instance, type CONTROL and FLASH ON, type a few letters. Now press ESC then O, type a few more letters and you will see that they no longer flash. The same thing happens if reverse field, insert and quote mode are on (quote mode is when you have opened quotation marks, and whenever you press a cursor control key a symbol appears instead).

When editing a program it can be useful to erase whole parts of lines. The ESC P and ESC Q functions allow you to do this. If you type ESC P then anything before the cursor will be deleted. Whatever is under the cursor will also be erased with both of these functions.

The display which your computer sends out to the television is sometimes too wide to fit on the screen. The best way to cure this problem is to use a different television set, but if you have not another one available you can reduce the size of the display by typing ESC and R. The screen will then

clear, leaving a border one character wide all round the screen. Pressing ESC then N will return the screen to its normal size.

You may, if you wish, use the ESC key to scroll the screen up or down. Pressing ESC then V will scroll the screen up, while pressing ESC then W will scroll the screen down.

If you press the ESC key by mistake then you should press the X key as this cancels the ESC function.

The ESC functions are probably at their most useful when used in a program. You have already used the ESC functions in this way to define the size of a window from BASIC, so you should know that to implement these functions you must PRINT CHR$(27) and then enclose the letter which you would normally type in quotation marks, so

```
PRINT CHR$(27);"W"
```

will scroll the screen down one line, in the same way as if you had pressed ESC then W.

These special functions can be extremely useful in programs, and in command mode. It would be a good idea to make sure that you know how to use them properly, as you will be surprised at how often they come in useful.

CHAPTER 7

We All Make Mistakes

Error trapping

For a computer program to be really good it must be 'user friendly', which means that whoever uses the program is given full instructions on how to use the program properly, and that whatever that user does, however stupid, the program should carry on working and tell the user what he or she has done wrong. To help you to write such programs your computer has been supplied with error trapping facilities, so that if, for example, you press the RUN/STOP key when you do not want to, then the program will be able to explain what you have done wrong.

Here is a short program which illustrates this

```
10 TRAP 90
20 SCNCLR
30 PRINT"PLEASE DO NOT PRESS THE RUN/STOP KEY"
40 PRINT"WHILE I FILL THE SCREEN WITH O's"
50 FOR N = 1 TO 500:NEXT N
60 FOR N = 1 TO 1024:PRINT"O";:NEXT N
70 FOR N = 1 TO 500:NEXT N
80 RUN
90 SCNCLR
100 IF ER = 30 THEN PRINT"I ASKED YOU NOT TO PRESS THAT
KEY!":ELSE STOP
110 PRINT"NOW, I WILL START AGAIN"
120 PRINT"BUT PLEASE DO NOT PRESS THE RUN/STOP KEY"
130 FOR N = 1 TO 2000:NEXT N
140 RESUME 20
```

Line 10 of this program contains the first of the error trapping commands — TRAP. This command tells the computer which line to jump to when it comes across an error. In this case the TRAP command tells the computer that if there is an error it should go to line 90 and carry on with the program from there.

The rest of the program is straightforward, until you come to line 100. In this line we refer to a variable, ER, which has not been assigned any value

by the computer. This variable is a system variable, which is a variable used by the computer. ER contains the number of the last error which was made. Since the error number for pressing the RUN/STOP key (pressing the RUN/STOP key is classed as an error by the computer) is 30, the computer will carry out all the instructions from lines 100 onwards only if the RUN/STOP key had been pressed.

Line 140 has another new command on it. The RESUME command allows the computer to continue with the main program after an error has occurred. RESUME is similar to GOTO except that it also marks the end of the error-trapping routine, so when the computer comes across line 140 it realises that this is the end of the error-trapping subroutine, and then jumps to line 20 to continue with the program from there.

There is another version of the RESUME statement — RESUME NEXT. This statement tells the computer to go to the main program and carry on with it from the NEXT statement after the one which caused the error. For instance, change line 140 to

```
140 RESUME NEXT
```

then type RUN. At some point press the RUN/STOP key. You will receive the same message telling you that you shouldn't have pressed that key, and then, after a short pause, the computer will continue with the program from where it was stopped.

There is another system variable which is used for errors. This variable is EL and contains the line number in which the last error occurred. So if you change line 10 to

```
10 HELLO
```

and RUN the program you will immediately get an error (obviously). If you now type

```
PRINT EL
```

the computer will display the number 10, which is the line in which the error occurred.

We know that the system variable ER contains the error number of the last error that occurred, but a number doesn't tell you much about what has happened. For instance, if you were told that error number 11 has just occurred you wouldn't be much the wiser. Fortunately, there is a function which helps us in this matter — ERR$. If you type

```
PRINT ERR$(11)
```

the computer will display the message SYNTAX on the screen. This is because error number 11 is a SYNTAX ERROR. Similarly, if you type

```
PRINT ERR$(14)
```

you will see the message ILLEGAL QUANTITY displayed on the screen. You can use the ERR$ to find the message for any error number except those which refer to the disk drives.

Summary

It is possible to TRAP an error so that whenever the computer comes across an error it will jump to your own routine which will deal with it.

The RESUME command marks the end of your error-handling routine and also tells the computer which line to jump to in order to carry on with the rest of the program.

The system variable ER contains the number of the last error which occurred.

The line number in which the last error occurred is contained in the system variable EL.

To find out the error message which goes with any error number you should use the ERR$ function.

HELP

The HELP command is extremely useful for when you are trying to find a mistake in a program line. If, for instance, you have a line with four or five commands in it, and you know that there is a mistake on that line but do not know where, you simply have to type HELP and the line with the mistake in it will appear on the screen. The actual command with the mistake will flash so that you can identify it easily. The HELP command will only work after you have received an error message (pressing the key marked HELP has the same effect as typing HELP).

TRON and TROFF

It is very rare when a program of any length or complexity works first time. Finding real errors (the ones which the computer spots as errors and tells you about) are no problem, especially with the HELP command. However, there are often bugs in the program which, although the program works, prevent the program from doing exactly what it is supposed to do.

In order to make the extermination of bugs much easier your computer

has been given two commands — TRON and TROFF. The TRON command tells the computer to turn the TRace ON. When this happens the computer will display the line number of the line which is currently being carried out on the screen, so as soon as you see the mistake you can look to see which line is being carried out, and that will be the line with the mistake in it.

For instance, if we wanted the message on line 20 of the following program to read SAUSAGE AND MASH instead of BEANS ON TOAST we could use the TRON command to find the line which displays the message BEANS ON TOAST (I know you can see it, but in a program with about 200 lines it would be a bit more difficult to spot, and this is an example). So, type in this program, type TRON and then RUN the program

```
10 SCNCLR
20 PRINT"BEANS ON TOAST"
30 FOR N = 1 TO 10
40 PRINT"THIS IS A TEST";
50 NEXT N
```

As each line is carried out you will see the line number appear enclosed in square brackets ([ ]). You will see [20] appear as the message BEANS ON TOAST appears on the screen, so you will immediately know that line 20 displays that message. The trace will remain on until you turn it off with the TROFF (TRace OFF) command.

CHAPTER 8

More Advanced Programming

READ, DATA and RESTORE

It is very often useful to be able to have a list of numbers or characters which the computer can refer to and use. These numbers or characters could be a list of names and addresses which you want the computer to search through for a specific name, for example. One way of doing this is to store each persons's name and address in a string variable, or possibly a string array, and search through each variable when you want to search for a person's name. An easier way, though, would be to have a list of DATA which the computer could look through for the correct name. To set up such a list we use the DATA statement, like this.

```
1000 DATA "FRED BLOGGS", 123456, "JOHN BROWN",123642
```

This is just a short list, only two names and telephone numbers, but the list could go on, using several DATA statements. As you can see from the above DATA statement, any characters can be included in the list, and can be enclosed in quotation marks, although this is not essential. Numbers can also be stored in the list, and both numbers and characters can be mixed freely, as you can see.

It is not much use having a list if you cannot do anything with it. What we have to do is READ each piece of data into a variable, the numbers being read into a numeric variable, and the other characters being read into a string variable (numbers can also be read into a string variable if you want, but you cannot perform calculations with them if you do this). This short program is a simple telephone directory which asks you for the person's name and then tells you their telephone number

```
10 SCNCLR:INPUT"WHAT IS THE PERSON'S NAME";NAME$
20 FOR N = 1 TO 5:READ A$,X
30 IF A$ = NAME$ THEN PRINT"THAT PERSON'S TELEPHONE
NUMBER IS";X:GOTO 60
40 NEXT N
```

```
50 PRINT"SORRY, I DON'T KNOW THAT PERSON'S TELEPHONE
NUMBER"
60 PRINT:PRINT"PRESS ANY KEY"
70 GETKEY A$:RUN
80 DATA FRED BLOGGS, 123456,JOHN BROWN,123642,BILL
SMITH,129832
90 DATA JACK JONES,126452,PETER JOHNSON,327428
```

When you RUN this program the screen will clear and you will be asked for the name of the person whose telephone number you want (choose one from the names in the DATA statements). The computer will then search through the list of DATA and if it finds that person's name it will tell you his telephone number (this is immediately after the name in the list). If the computer does not find the person's name then it will tell you.

The program works like this

Line 10: Clear the screen. Display the message 'WHAT IS THE PERSON'S NAME' and then wait for a response before assigning that response to the string variable NAME$
Line 20: Start repeating everything between the FOR and NEXT command 5 times, with the value of the variable N starting at 1 and increasing by 1 each time round the loop until it reaches 5. READ the next piece of DATA from the list and assign it to the string variable A$, then READ the next peice of DATA and assign it to the variable X
Line 30: Test to see if the string variable A$ is the same as the string variable NAME$. If it is then display the message 'THAT PERSON'S TELEPHONE NUMBER IS' and then display the value of the variable X before jumping to line 60 and carrying on with the program from there
Line 40: Marks the end of the FOR. . . NEXT loop
Line 50: Display the message 'SORRY, I DON'T KNOW THAT PERSON'S TELEPHONE NUMBER'
Line 60: Display a blank line then display the message 'PRESS ANY KEY'
Line 70: Wait for a key to be pressed and assign the symbol on that key to the string variable A$ before re-starting the program
Line 80: List of DATA
Line 90: List of DATA

Line 20 of the program READs in the next piece of DATA from the list at the end of the program (in fact it READs two pieces of DATA, but only one at a time). Each time a piece of DATA is read in the computer remembers where the next piece of DATA is, so that the next time it comes across a READ statement it knows where to take the DATA from.

Several pieces of DATA can be read in with one READ statement, as you can see from the above program. All that you have to do is tell the computer which variables you want the DATA to be assigned to in the correct order, and separate each variable with a comma. You can READ in as many pieces of DATA as you like, as long as the variable list will fit onto a normal program line, if it doesn't you will need to use two lines and two READ statements.

But what happens when the computer reaches the end of the DATA list? Well, once the computer reaches the end of the list it has no more DATA to READ in, so it gives you an OUT OF DATA ERROR if you keep on trying to READ in more DATA. For instance, if you change line 70 of the above program to

```
70 GETKEY A$:GOTO 10
```

and RUN the program for a while you will eventually receive an OUT OF DATA ERROR. This is because the RUN command (which was originally used in line 70) also resets the DATA pointer (the one which the computer uses to remember where it is in a list of DATA) back to the start of the list, whereas a GOTO command does not. To overcome this problem, we use the RESTORE command, which tells the computer to start READing in DATA from the beginning of the list again. If you now change line 70 to

```
70 GETKEY A$:RESTORE:GOTO 10
```

then the program will work perfectly.

You may also tell the computer to start taking DATA from a certain line number. For instance, alter line 70 to

```
70 GETKEY A$: RESTORE 90:GOTO 10
```

The RESTORE 90 statement tells the computer to set the data pointer to the start of the DATA in line 90. You will find that you will only be able to find the addresses of Jack Jones and Peter Johnson.

You may have thought up to now that when the computer reaches a READ statement it jumps to the line which has the DATA on it and looks through that line for the next piece of DATA. This is not the case, however, and to prove this type TRON and RUN the program. You will see that at no time does computer jump to lines 80 or 90 to look through the DATA list. This is because the computer knows exactly where the next piece of DATA is stored in it memory, so it does not need to bother about line numbers, it goes straight there.

Summary

The DATA statement is used to store lists of characters and numbers. The characters can be enclosed in quotation marks but this is not essential.

The READ statement is used to READ in a piece of DATA assign it to a variable. This statement can be used to READ in several pieces of DATA at once and assign each piece to a separate variable. In this case the variables in the list following the READ statement must be separated by commas.

The RESTORE statement tells the computer to go back to the beginning of the DATA list and start READING in DATA from there.

The computer does not actually go to the line which contains the DATA when it is told to READ in a piece of DATA — it knows exactly where the DATA is stored and does not need to bother with line numbers.

String handling

String variables are extremely useful, and there are many ways in which you can chop them up and re-arrange them to suit your needs. The commands for manipulating strings variables are covered in the next few pages.

LEFT$

If you can remember the chapter on IF...THEN...ELSE you may recall three of four lines similar to this

```
130 IF A$ = "YES" OR A$ = "Y" THEN...
```

It would be much easier if we could test to see if the first letter of A$ is a Y because then we could accept answers such as Y, YES, YEAH, or virtually any other version of the word YES starting with a Y. To do this we use the LEFT$ command, like this

```
130 IF LEFT$(A$,1) = "Y" THEN...
```

This particular line tests to see what the first character of the variable A$ is. If you change line 130 to

```
130 IF LEFT$(A$,2) = "YE" THEN...
```

then the computer will test to see what the first two characters of the variable A$ are (the 2 in the LEFT$ command tells the computer you want the first 2 characters, changing it to 3 would mean the first 3 characters).

Of course, you don't have to use a string variable with the LEFT$

command (or any of the other string handling commands, come to that), you could also use characters enclosed in quotation marks, like this

```
250 IF LEFT$("COMPUTER",4) = "COMP" THEN...
```

Here is a short example program

```
10 SCNCLR:INPUT"DO YOU LIKE USING COMPUTERS";CO$
20 IF LEFT$(CO$,1) = "Y" THEN PRINT"I'M GLAD ABOUT THAT!"
30 IF LEFT$(CO$,1) = "N" THEN PRINT"OH, I'M NOT TO BLAME!"
```

RIGHT$

The RIGHT$ command is very similar to LEFT$, except that LEFT$ tests to see what the first characters of a string are, but the RIGHT$ command looks to see what the last characters of a string are. Try this example

```
10 SCNCLR:INPUT"TYPE IN SOMETHING PLEASE";ZZ$
20 PRINT"THE FIRST 2 CHARACTERS THAT YOU TYPED IN
WERE";
30 PRINT LEFT$(ZZ$,2):PRINT"THE LAST 2 CHARACTERS THAT
YOU TYPED IN WERE";
40 PRINT RIGHT$(ZZ$,2)
```

The 2 in the RIGHT$ command tells the computer that you want the last two characters of the string, and can easily be changed, just like the LEFT$ command.

MID$

The MID$ command is used to find out what the middle characters are, rather than the beginning or end ones. Rather than just saying how many characters you want to test, as you do with the LEFT$ and RIGHT$ commands, you also have to say where you want to start from. For example, if you had a line such as this

```
310 A$ = MID$(B$,4,3)
```

then the computer will assign three characters from the middle of the string variable B$ to the string variable A$, with the first of these characters being the fourth character of the string variable B$.

It is also possible to replace parts of a string variable by using the MID$ command. Here is a short example program which does this

```
10 SCNCLR
20 A$ = "HELLO HELLO EVERYBODY!"
30 PRINT A$
40 MID$(A$,7,5) = "THERE"
50 PRINT A$
```

If you look at line 40 you will see the MID$ command being used to replace the second HELLO with THERE. This is done simply by telling the computer where in the string variable you want to start (the seventh character in this case) and how many characters you want to replace (in this example we want to replace five characters) and then tell the computer which characters you want to replace the old ones with. As you can see the replacement characters must be enclosed in quotation marks.

INSTR

The INSTRing command is used to find out if one string is contained IN another STRing. Try this short program

```
10 SCNCLR:A$ = "PETER PIPER PICKED A PECK OF PICKLED
PEPPERS"
20 PRINT INSTR(A$,"PICK")
```

When you RUN this program the number 13 will appear on the screen, because the P of the letters PICK is the 13th letter of the string variable A$. What you have just told the computer to do is to search through the string variable A$ to see if the letters PICK are contained within it. If these letters are contained in A$ then the computer will tell you exactly where the first letter of PICK appears in the string A$.

If you look at the string A$ you will see that the letters PICK appear twice, once in the word PICKED and once in the word PICKLED. The computer will only find the first occurrence of the characters which you are searching for. In order to make it find the second occurrence of the letters PICK you will have to change line 20 to

```
20 PRINT INSTR(A$,"PICK",15)
```

When you RUN the program this time the number 30 will appear on the screen. This time the computer has started searching for the letters PICK at the 15th character of the string A$. In other words, the computer will search through the letters CKED A PECK OF PICKLED PEPPERS for the first occurrence of the letters PICK, and find that the P of PICK is the 30th character of the string variable A$.

If the computer cannot find the characters that you are searching for in the string then it will return the number 0.

LEN

The LEN command is used to find out the LENgth of a string variable, or how many characters it contains. This short program illustrates the use of the command

```
10 SCNCLR:A$ = "SUPERCALIFRAGILISTICEXPIALIDOTIOUS!"
20 PRINT"THE STRING VARIABLE 'A$' CONTAINS ";
30 PRINT LEN(A$);"CHARACTERS!"
```

As you can see, the string variable that you want to refer to must be enclosed in brackets after the LEN command.

Summary

The LEFT$ command is used to obtain the first few characters of a string variable.

The RIGHT$ command is used to obtain the last few characters of a string variable.

The MID$ command is used to obtain characters from the middle of a string variable.

The INSTR command is used to find out if one string is contained within another. This command will return the position of the first character of the string which you are searching for in the string that you are searching through. If the string that you are searching is not contained in the other string then the number 0 will be returned.

The LEN command is used to find the LENgth of a string variable, which must be enclosed in brackets after the LEN command.

Sound and Volume

It is time we explored your computer's sound capabilities. Your computer has three different 'voices', and you can use any two of these together. This means that you could have one voice playing a tune and another voice playing the rhythm. Two of the voices produce tones and the other produces white noise (this is useful for explosions and gun shots).

However, before we can produce any sound we have to set up the volume level. To do this we use the VOL command, together with a number from 0 to 8 (8 is maximum VOLume, 0 is minimum, or off). It is best to set the VOLume to maximum, so type in

VOL8

Now all we have to do is to choose a voice and a note and decide how long to play it for. For instance, if we wanted to play the note C on voice one for three seconds then we would type the command

SOUND 1,810,180

The number 1 tells the computer that we want to use voice one, the number 810 tells it that we want to play the note C in the third octave, and the number 180 tells the computer that we want the note to last 3 seconds (the length of the note is in sixtieths of a second).

Try changing the 1 of the SOUND command to a 2 and then a 3 to see what the different voices sound like.

As I said before, it is possible to have two voices playing at once. You can either have voices 1 and 2 playing together, or voices 1 and 3. If you type this line you will hear a note being played over the top of white noise

SOUND 1,810,360:SOUND 3,917,360

The SOUND command can be used to play tunes and to create sound effects. Here are a few programs which do this

Doctor Foster

```
10 SCNCLR:VOL 8:PRINT"DOCTOR FOSTER"
20 FOR N = 1 TO 36:READ NOTE, LENGTH
30 SOUND 1, NOTE,LENGTH
40 NEXT N
50 FOR N = 1 TO 5000:NEXT N
60 RUN
70 DATA 739,60,739,30,834,60,834,30,810,30,834,30,810,30,798,60,
770,30
80 DATA 739,60,739,30,798,30,770,30,739,30,770,90,770,60,770,30,
739,30
90 DATA 739,30,739,30,834,30,810,30,798,30,810,30,834,30,810,30,
854,30
100 DATA 834,30,810,30,798,30,798,30,798,30,880,60,770,30,881,90,
881,60
```

Telephone

```
10 SCNCLR
20 VOL8
30 FOR M = 1 TO 10
40 FOR N = 1 TO 10:SOUND 1,650,1:SOUND 1,700,1:NEXT N
```

```
50 FOR N = 1 TO 50:NEXT N: FOR N = 1 TO 10:SOUND 1,650,1:
SOUND 1,700,1:NEXT N
60 FOR N = 1 TO 1000:NEXT N,M
70 FOR N = 1 TO 8:SOUND 1,650,1:SOUND 1,700,1:NEXT N
80 SOUND 1,920,20
90 FOR N = 8 TO 0 STEP -1:VOL N:FOR M = 1 TO 5:NEXT M,N
```

The computer can continue to carry out other instructions while making a sound, so you may have a tune playing in the background while something else is happending.

Summary

The volume level is selected by the VOL statement. The level should be from 0 to 8.

There are three voices available, and voices 1 and 2, or voices 1 and 3 may be used simultaneoulsy.

A note can be of any length from 0 to 65535 sixtieths of a second.

There are 1024 possible sounds which may be produced. See the table on the following page for the number of each musical note.

NOTE	FREQUENCY (Hz)	VALUE
A	110	11
B	123.5	122
C	130.8	173
D	146.8	266
E	164.7	349
F	174.5	387
G	195.9	457
A	220.2	520
B	246.9	575
C	261.4	600
D	293.6	647
E	330	689
F	349.6	708
G	392.5	743
A	440.4	774
B	494.9	802
C	522.7	814
D	588.7	838
E	658	858
F	699	868
G	782.2	885

A	880.7	901
B	989.9	915
C	1045	921
D	1177	933
E	1316	943
F	1398	948
G	1575	957

This table covers four octaves, but sharps and flats are not shown. The frequency of the note is given for reference only. The value shown for each note are the ones which you should use as the second number after the SOUND statement. For example, to play the note C (third note down) for half a second you would use the command

SOUND 1,173,60

You may play a note of nearly any frequency. If you know the frequency of the note which you require then you can calculate the value to be used in the SOUND statement using this formula

VALUE = 1024 – (110840.45/FREQUENCY)

ON...GOTO and ON...GOSUB

In the chapter on GOTO and GOSUB you were told that you cannot use a variable instead of a line number with these commands. To make up for this, your computer has been supplied with the commands ON...GOTO and ON...GOSUB. These commands will GOTO or GOSUB a line depending on the value of a variable. For instance, if the computer encountered this line

120 ON ZZ GOTO 1000,2000,3000,4000

then it would look to see what number the variable ZZ represents and then GOTO one of the following line numbers depending on that value. If the value of ZZ is 1 then the computer will GOTO line 1000. If the value is 2 then the computer will GOTO line 2000, and so on.

The ON...GOSUB command works in exactly the same way as the ON...GOTO command, except that it GOes to a SUBroutine which should end with a RETURN statement, as with a normal GOSUB command.

Here is a short program which simulates a die and uses the ON...GOSUB command, for example

```
10 SCNCLR
20 FOR N=9 TO 13:CHAR 1,17,N,"[CONTROL 5 CONTROL 9]
      ":NEXT
30 DICE = INT(RND(0)*6) + 1:IF DICE = 7 THEN GOTO 30
40 ON DICE GOSUB 60,70,90,110,130,150
50 FOR N = 1 TO 1000:NEXT:RUN
60 CHAR 1,19,11,"*":CHAR 1,17,14," ONE ":RETURN
70 CHAR 1,18,10,"*":CHAR 1,20,12,"*"
80 CHAR 1,17,14," TWO ":RETURN
90 CHAR 1,18,10,"*":CHAR 1,19,11,"*":CHAR 1,20,12,"*"
100 CHAR 1,17,14,"THREE":RETURN
110 CHAR 1,18,10,"* *":CHAR 1,18,12,"* *"
120 CHAR 1,17,14,"FOUR ":RETURN
130 CHAR 1,18,10,"* *":CHAR 1,19,11,"*":CHAR 1,18,12,"* *"
140 CHAR 1,17,14,"FIVE ":RETURN
150 CHAR 1,18,10,"* *":CHAR 1,18,11,"* *":CHAR 1,18,12,"* *"
160 CHAR 1,17,14," SIX ":RETURN
```

Where you see the square brackets in Line 20, this indicates that you should change the text colour to purple and switch reverse on by pressing CONTROL 5 then CONTROL 9.

The program works in this way

Line 10: Clear the screen
Line 20: Change the text colour to purple and print a solid block 5 lines deep and 5 characters wide, with the first block in the character square 17 across and N characters down
Line 30: Choose a random number between 0 and 1, multiply it by six and round it up before adding one to the result. Assign the final number to the variable DICE. If the value of the variable DICE is seven then carry out this line again (choosing a random number between 1 and 6, then adding 1 and rounding down makes it possible that a six will come up. The computer very rarely chooses the highest number possible when choosing a random number and therefore RND(0) * 6 is almost always less than seven)
Line 40: If the value of the variable DICE is 1 THEN GOTO the subroutine starting at line 60. If the value of DICE is 2 THEN GOTO the subroutine starting at line 70. If the value of DICE is 3 THEN GOTO the subroutine starting at line 90, and so on
Line 50: Empty FOR. . . NEXT loop — causes a delay before re-running the program
Line 60: Display a single star at the character square 19 across and 11 down then display the message ONE with the 0 in the 17th column across and the

14th row down before returning to the command immediately after the GOSUB command which jumped to this routine

Line 70: Display a single star in the character square 18 across and 10 down then display another star in the character square which is 20 across and 12 down

Line 80: Display the message TWO with the T in the 17th column across and 14th row down

All lines after line 80 are similar to 60-80 — they just display various numbers of stars.

AUTO

If you are typing in a program from a book or a magazine then it can become very boring having to type line numbers, especially if the lines are numbered evenly (e.g. 10, 20, 30 and so on). In order to speed up the entry of such programs, and also to make it less boring, your computer has been equipped with an AUTOmatic line number command. If you type in

```
AUTO 10
```

and then start typing in a short program you will see that as soon as you have pressed RETURN at the end of the first line, the next line number will appear for you AUTOmatically.

The number after the AUTO command tells the computer how much each line number is increased by, so if you type AUTO 50 the computer will number the lines 50, 100, 150 and so on.

Once you come to the end of the program you should press the RETURN key without typing anything. For instance, if you have finished your program at line 1510 and the computer displays the next line number 1520 and waits for you to type in some more of your program you should just press RETURN without typing anything else.

CLR

In some circumstances you may need to reset all the variables in the middle of a program. The easiest way to do this is to use the CLR command. The program itself is not stopped or altered in any way.

The CLR command is carried out automatically when you alter a program line, or RUN a program.

ASC

It is often useful to be able to find out the CHR$ code for a character. Fortunately for us, your computer has a command which allows us to find

out the CHR$ code for any character without having to look it up in a table. This command is ASC.

If you type

```
PRINT ASC("A")
```

You will see the number 65 appear on the screen. 65 is the CHR$ code for the A symbol. You can find the CHR$ code for any character like this, all you have to do is enclose the character in quotation marks, and enclose them in brackets, as in the above example.

VAL

The VAL command is a function which returns the value of a string variable. For instance, if the string variable BS$ had previously been set to 921 then the command

```
Z = VAL(BS$)
```

would assign the number 921 to the variable Z.

If there is a combination of letters and numbers in the string variable then one of two things will happen. If the string starts with a number then the value returned by the VAL command will be the value of that number (eg PRINT VAL("A12") would return the value 0).

STR$

STR$ is the opposite to VAL, for this function converts a number to a string. For instance, if you had a line such as this

```
320 A$ = STR$(864)
```

then the string variable A$ would be assigned the characters 864.

The STR$ command will always add a space before the number at the start of the string variable in which the characters are to be stored. For instance, if you entered this program

```
10 SCNCLR:A$ = STR$(864)
20 PRINT A$:PRINT LEN(A$)
30 A$ = RIGHT$(A$,3)
40 PRINT A$:PRINT LEN(A$)
```

You will see that a space has been added before the number when it was stored in the string variable A$. Line 30 effectively removes this space, and when the string is displayed a second time along with the number of characters in that string, you will see that the space has been removed.

CHAPTER 9

Printing and Graphics

Print using

PRINT USING is a version of PRINT which is extremely useful for displaying tables and charts.

Suppose, for example, that you want a maximum of six digits in each column of a table. To do this you would use a line like this

```
760 PRINT USING" # # # # # # ";NU
```

This line would round up the value of NU (if it had a decimal point in it) and add spaces before the number until the total length of digits and spaces has six characters. For instance, if the value of NU was 35.6 then the display would look like this (the ' + ' symbol represents a space — you will not see it displayed on the screen).

```
+ + + + 37
```

The number has been rounded up, and four spaces have been added before it, so with the two digits (3 and 7) the total number of characters is 6, which is exactly the number of hash symbols (#) that you used in the PRINT USING statement. If you had used seven hash symbols then five spaces would have been added before the number.

If the value of NU was a number containing more than 6 digits then the computer would have displayed six stars instead of the number, indicating that the number was too long.

The hash symbols can also be used to define how many characters of a string you want displayed. For instance, if you type

```
PRINT USING" # # # # ";"ABCDEFG"
```

then only the characters ABCD will be displayed, because you have told the computer that you only want four characters to be displayed.

You can also tell the computer to indicate whether a number is positive or negative and where to put the + or − sign. For instance, if you type

```
PRINT USING" + # # # # #";4325
```

then the computer will display the number 4325 with a plus syumbol before it. If the number 4325 had been negative then a minus sign would have been displayed before the number instead (try changing 4325 to a minus number).

If you now type

```
PRINT USING" # # # # + ";4325
```

then you will see the computer display the number 4325 with the plus symbol *after* it.

If you try the two previous examples with a minus sign instead of a plus sign, like this

```
PRINT USING" - # # # # #";4325
```

```
PRINT USING" # # # # - ";4325
```

and with positive and negative numbers then you will see that if you use a minus sign then the computer will display a minus sign before or after a number (depending on where you tell the computer to put it) if the number is negative, but will not display a plus sign if the number is positive.

It is also possible to tell the computer how many numbers you want before and after the decimal point. For instance, if you type

```
PRINT USING" # # # # # . # # # # #";143.65786
```

then you will see the computer round up the number 143.65789 to four decimal places and put two spaces (represented here by plus signs) before the number, like this

```
+ +143.6579
```

This is because we have told the computer that we want four digits after the decimal point, so it has to round the number up slightly. We have also told the computer that we want five digits before the decimal point, and because we have given it only three digits before the decimal point it has to add two spaces.

It is sometimes useful to be able to add commas in a number for clarity. For instance, 1,000,000 is much clearer than 1000000. We can use the PRINT USING command to tell the computer to add commas in the correct places when displaying a number. Try this example

```
PRINT USING" # #,# # #,# # #";1000000
```

the computer will display the result

+ 1,000,000

If you change the number to 1000 then the computer will display

+ + + + + 1,000

This is because we have told the computer to display a total of seven characters (that is why the spaces are added) not including the commas, and to add commas before every third digit from the left (e.g. the comma in 1,000 is before the third digit from the left, and the comma in 1,000,000 is before the sixth digit from the left).

The PRINT USING command is also useful when money is being indicated. Try this example

PRINT USING"$ # # # # # # # #";1234

You will see this result displayed on the screen

$ + + + 1234

However, if you type

PRINT USING" # $ # # # # # # #";1234

then you will see this result

+ + + $1234

Putting a dollar symbol after the first hash symbol tells the computer that you want the dollar symbol to float. This means that the dollar symbol will always be displayed immediately before the first digit of a number. The first hash symbol is still taken into account by the computer (in the above example we had one hash symbol before the dollar symbol, and six after, and the computer treated these as seven hash symbols as if the dollar symbol was not there). If the dollar symbol is put before the first hash symbol then it will be displayed at the far left of the number, so that any spaces that the computer adds will go between the dollar symbol and the number.

If the need arises to display a number in exponential form (e.g. 3E + 04 is 3 times ten to the power of four, and 7E – 02 is 7 times ten to the power of minus two), then the PRINT USING command can cater for it. All you have to do is add four ↑ symbols at the end of the PRINT USING statement, like this

```
PRINT USING " #↑↑↑↑";14326
```

This command will display the result

```
1E+04
```

which is 1 times ten to the power of four, or 10000. The computer has rounded down the number 14326 to 10000. If you instead typed in

```
PRINT USING " # #↑↑↑↑";14326
```

you will get the result

```
14E+03
```

which is 14 times ten to the power of three, which is 14000.

Another useful feature of the PRINT USING command is the ability to centre a string in a field. If you had a column in a table that was six characters wide, and you wanted to centre the letters AA in that column, then the computer would have to display two spaces, then AA then another two spaces. Type

```
PRINT USING" # # # # # # =";"AA"
```

then the computer would display this

```
++AA++
```

Of course, you will not actually see the spaces (represented by plus signs) after the letters AA. We used six hash symbols in the PRINT USING command, and this tells the computer that you want the field to be six characters wide. The = symbol tells the computer that we want the characters following to be centred.

Finally, if you want to display a string flush to the right of the field, then you would use a command like this

```
PRINT USING" # # # # #>";"AA"
```

which would produce this result

```
+++AA
```

We have told the computer to display the characters AA flush with the righthand side of a field which is 5 characters wide.

Summary

The PRINT USING command can be used in many ways to help display tables, and generally to tidy up the screen display. It is possible to

(i) Define how many characters or digits you want to display in a field (one hash symbol represents one character).

(ii) Define whether you want a plus or minus symbol after or before a number (by placing a plus or minus symbol before or after the hash symbols).

(iii) Specify how many digits you want before and after the decimal point (by putting a decimal point in the corrct place between the hash symbols).

(iv) Specify where commas should occur in a number (by placing commas in the correct place between the hash symbols).

(v) Tell the computer to put a dollar symbol before a number, either floating or fixed by placing a dollar symbol either before the hash symbols (fixed) or after the first hash symbol (floating).

(vi) Display a number in exponential form by adding four ↑ signs after the hash symbols.

(vii) Centre a string in a field by using the = symbol after the hash symbols.

(viii) Display a string flush with the right of a field by using the > symbol after the hash symbols.

Of course, wherever in this chapter we have used strings and numbers you can use variables, and wherever we used variables you can use strings or numbers. The PRINT USING command can be used both in command mode and in programs.

PUDEF

The PUDEF command is used to alter the way in which the PRINT USING command displays numbers and characters. For instance, if you wanted the computer to display stars instead of spaces then you would use this command

PUDEF"*"

If you then had the command

PRINT USING" # # # # # ";12

Then the computer would display the result

***12

In some cases you may wish for a hyphen to be displayed instead of a comma, and to allow this you would use

PUDEF" – "

Or, if you wanted a decimal point instead of a comma then you would use

PUDEF" ."

You may need to display a / symbol instead of a decimal point, in which case you would use

PUDEF" ,/"

The most useful function of the PUDEF command is to make the computer display a pound symbol instead of a dollar symbol, like this

PUDEF" ,.£"

You can change one or all of the symbols ' ',',''.' and '$' to any symbol you like using the PUDEF command. You may have noticed already that the first character of PUDEF command is the character that will replace the space, the second character is the one that will replace the comma, the third character is the one that will replace the decimal point, and the last character replaces the dollar symbol. This means that to set everything back to normal you would use

PUDEF" ,.$"

But if you wanted to change the decimal point to a '%' sign you would use

PUDEF" ,%"

with the '%' symbol in the position normally occupied by the decimal point.

Graphics

Until now your programs have been somewhat limited in that they have not

been able to use the excellent colour and graphics capabilities of your computer. You will probably know that your computer is capable of displaying 121 colours and can produce some very good pictures, and now is the time to find out how it is done.

COLOR

The first thing you may think is that I have mis-spelt the subtitle, but because your computer works in a language which was invented in America, it uses the American spellings for most commands. This is generally the case with most computers.

The COLOR command can be used to change the colour of the screen, the background and what is put on the screen. If you type

```
COLOR 0,15,6
```

the whole screen will become the same colour.

The first number of the COLOR command tells the computer what you want to change the colour of. The number can be from 0 to 5, and represents the following

0 — SCREEN COLOUR
1 — TEXT COLOUR
2 — MULTICOLOUR 1
3 — MULTICOLOUR 2
4 — BORDER COLOUR

You have not come across multicolour mode yet, but when you do you will need to know that the COLOR command is used to choose the colours.

The second number is the actual colour that you want. This is a number between 1 and 16 — 1 is black, 2 is white, 3 is red and so on, the number of each colour corresponds to the number on the key which that colour is written on. The numbers of the colours on the bottoms of the keys are found by adding 8 to the numbers on those keys (e.g. pink has the number 12, which is 8 + 4).

The third number is the luminance, or brightness, level. This is a number ranging from 0 to 7, with 0 being the darkest, and 7 being the brightest. The luminance level has no effect on black. The fact that you can choose how bright you want each colour to be gives the impression that there are more colours than there really are, so although you only have sixteen colours, it looks like you have 121.

In order to see the full range of colours which your computer can produce, type in this short program

```
10 SCNCLR
20 FOR C = 1 to 16: FOR I = 0 TO 7
30 COLOR 1,C,I:PRINT "[RYS ON][RYS OFF]";
40 NEXT I,C
```

This program will change the text colour through all the possible colours and luminance levels and display a coloured block in each of these colours. If you want to see the screen and border in all the available colours then you should change line 30 to

```
30 COLOR 0,C,I:COLOR 4,C,I:FOR M = 1 TO 500:NEXT M
```

GRAPHIC

The GRAPHIC command is used to choose which graphics mode we want to work in. There are five different modes, and each has its advantages and disadvantages. These modes are

GRAPHIC 0:	Normal text screen
GRAPHIC 1:	Normal high-resolution screen with 320 dots across and 200 dots down
GRAPHIC 2:	Similar to graphic 1 except has space for five lines of normal text at the bottom of the screen. The resolution is 320 dots across and 180 dots down
GRAPHIC 3:	Multicolour graphics. This allows more colours in a small area and has a resolution of 160 dots across and 200 dots down
GRAPHIC 4:	Similar to graphic 3 except it has five lines of normal text at the bottom of the screen. The resolution is 160 dots across and 180 dots down

A typical GRAPHIC command would be

```
10 GRAPHIC 2,1
```

Adding ',1' to the end of the GRAPHIC command has the effect of choosing the graphics mode and clearing the screen. Leaving the ',1' off or putting ',0' at the end selects the desired graphics mode but does not clear the screen.

Using graphics takes up a lot of the memory, but when you have finished using graphics you can gain access to this memory again by typing

```
GRAPHIC CLR
```

This command allows you to use the memory which the computer reserves whenever you use graphics.

LOCATE

The high-resolution screen has a special kind of cursor, called a pixel cursor, which is invisible to us, but which tells the computer exactly where it should be drawing on the screen. Although we cannot see this cursor, we can move it to any position on the screen. To do this we must tell the computer how many dots, or pixels across we want the pixel cursor, and how many pixels down, like this

```
175 LOCATE 40,30
```

This line tells the computer to LOCATE the pixel cursor at the point which is 40 pixels across from the lefthand side of the screen, and 30 pixels down from the top of the screen.

We can also tell the computer to move the pixel cursor by a given number of pixels up, down, left or right, like this

```
240 LOCATE + 10, – 5
```

This line tells the computer to move the pixel cursor by 10 pixels to the right and 5 pixels up from its current position. If we wanted the pixel cursor to move 10 pixels to the left and 5 pixels down then we would use a line like this

```
255 LOCATE – 10, + 5
```

You can also tell the computer to move the pixel cursor a certain number of pixels in a direction at a certain angle. The angle is measured from the vertical, like a compass bearing.

```
354 LOCATE 40;45
```

will move the pixel cursor by 40 pixels at the angle of 45 degrees — north-east using the compass analogy.

DRAW

The DRAW command allows you to draw on the screen. You can draw a single dot, you can draw a line, or you can draw several lines. Try this short program

```
10 GRAPHIC 1,1:COLOR 0,1,7:COLOR 4,7,7,:COLOR 1,2,7
20 DRAW 1,10,10
```

When you RUN this program you will see a white dot appear on the screen, which will be black with a blue border.

Line 10 selects graphics mode 1 and clears the screen. It then sets the screen colour to black, the border colour to yellow, and the draw colour to white.

If you look at line 20 you will see that we have given the DRAW command three numbers. The first is the colour source. This can be any number from 0 to 3. Selecting 0 tells the computer to DRAW in the background colour, which has the effect of rubbing out anything that it DRAWs over. Selecting 1 tells the computer to DRAW in the current text colour (which, in this case, is white). Selecting a 2 tells the computer to DRAW in multicolour 1, and selecting a 3 tells the computer to draw in multicolour 2. Since this program does not use a multicolour graphics mode, selecting a 2 or a 3 will cause the computer to draw in the colour that has been assigned to that colour source (ie if multicolour 1 had been set to 3 then selecting that colour source would make the computer draw in red).

The second number tells the computer how many pixels across you want to start drawing from, and the third number tells the computer how many pixels down you want to start drawing from. We have told the computer to start drawing from the point 10 pixels across and 10 pixels down, and since we have not told the computer to draw anything else it will just draw a single pixel at that location.

If you look at the dot on the screen you will see how small a pixel is, for that dot is one pixel. When you think that (in GRAPHIC 0 at least) there are 320 of those pixels across the screen, and 200 down, you will realise how much detail you can get in a picture.

The pixel on the screen will remain there until you tell the computer to return to the normal text screen. The easiest way to do this is to type any character (except a number) and then press RETURN. The screen will return to normal and you will receive an error message, but no harm will be done.

If you now alter line 20 to this

```
20 DRAW 1,0,0 TO 319,199
```

and RUN the program you will see a white line draw diagonally across the screen from the top lefthand corner to the bottom righthand corner.

What we have now told the computer to do is to DRAW a line from the point 0 pixels across and 0 pixels down, TO the point 320 pixels across and 200 pixels down. We can carry on adding to the DRAW command to produce something like this

```
20 DRAW 1,0,0 TO 319,199 TO 319,0 TO 0,199 TO 0,0
```

This will draw two triangles on the screen with their points meeting at the centre.

You can also tell the computer to DRAW a line from the pixel cursor to a point anywhere on the screen. For instance, if you change line 20 to

```
20 LOCATE 10,10:DRAW 1 TO 50,50
```

and RUN the program the cursor will position the pixel cursor at the point 10 pixels across and 10 pixels down, and then DRAW a line from that point TO the point 50 pixels across and 50 pixels down in the current text colour.

You can also DRAW from a point to a point so many pixels left or right of the start point, and so many pixels above or below the start point, like this

```
20 LOCATE 10,10:DRAW 1 TO +20,+10 TO -10,-5
```

When you RUN the program this time you will see that the computer has DRAWn a line from the pixel cursor to the point 20 pixels to the right and 10 pixels down from the pixel cursor, and another line from that point to the point 10 pixels to the left and 5 pixels above it.

The DRAW command can also DRAW lines at an angle. For instance, if you change line 20 to

```
20 LOCATE 50,50:DRAW 1 TO 40;32
```

and RUN the program, you will see the computer DRAW a line 40 pixels long at an angle of 32 degrees from the pixel cursor.

BOX

The BOX command is used to draw squares and rectangles. To see how it is used change line 20 of the above program to

```
20 BOX 1,110,50,210,150
```

and RUN the program this time you will see a square appear in the middle of the screen. This is because we have told the computer to draw a BOX with the top lefthand corner at the point 110 pixels across and 50 pixels down, and the bottom righthand corner at the point 210 pixels across and 150 pixels down. The computer then works out where the other two corners should be and draws a box (the first number is the colour source, as in the DRAW statement).

It is also possible to rotate a box. For instance, if you add ',45' to the end of the BOX command on line 20, like this

```
20 BOX 1,110,50,210,150,45
```

and RUN the program you will see drawn on the screen a square which has been rotated through an angle of 45 degrees. The angle can be anything from 0 to 360 degrees (with an angle of 360 degrees being the same as 0 degrees, which is no rotation at all).

You can also colour the BOX in. To do this you simply have to add ',1' after the angle of rotation (or just ',,1' if you do not want to rotate the BOX), like this

```
20 BOX 1,110,50,210,150,45,1
```

or

```
20 BOX 1,110,50,210,150,,1
```

The ',1' at the end of the BOX statement tells the computer to colour the box in in the current text colour (the colour which the box outline is drawn in).

CIRCLE

The CIRCLE command, as you will probably have guessed, is used to draw circles, but can also draw ellipses, triangles, squares and virtually any other shape, as you will soon see.

If you try changing line 20 of the above program to

```
20 CIRCLE 1,160,100,80
```

you will see a circle of radius 80 pixels appear on the screen, with the centre of the circle at the centre of the screen.

The first number is the colour source, and the second number is the number of pixels across you want the centre of the circle. The third number is the number of pixels down you want the centre of the circle. The last number is the radius of the circle.

If you now add ',40' to the end of the CIRCLE command, like this:-

```
20 CIRCLE 1,160,100,80,40
```

and RUN the program you will see an ellipse drawn on the screen, which is half as high as it is wide. This is because the number 80 (the radius) is actually the X radius, or half the width of the ellipse in pixels. The ',40' which we added is the Y radius, or half the height of the ellipse in pixels. If the X and Y radius are equal then the shape will be a circle, but if the X radius is larger then the ellipse is wider than it is tall.

It is also possible to draw an arc. If you alter line 20 as shown below then you will see an arc drawn

```
20 CIRCLE 1,160,100,80,80,90,180
```

The last two numbers which we have added tell the computer the start and end angles of the arc. The ',90' tells the computer to start drawing the arc from the angle 90 degrees, and the ',180' tells the computer to end the arc at the angle 180 degrees.

It is also possible to rotate your circle or arc (this may seem to be pointless, but you will see how it can be useful in a moment). To do this you simply have to add the angle of rotation to the end of the CIRCLE command, like this

```
20 CIRCLE 1,160,100,80,80,90,180,20
```

The program will now draw the same arc, but will be rotated by 20 degrees.

Finally, you can use the CIRCLE command to draw other shapes, such as triangles and hexagons. To do this you must tell the computer how many degrees you want between each side of the shape (this angle should be 360 divided by the number of sides on the shape you want). For instance, if you add ',120' to the end of line 20 a triangle will be drawn

```
20 CIRCLE 1,160,100,80,80,0,0,0,120
```

SCALE

SCALE is a simple command — it can either be on or off. To turn on the SCALE facility you should type SCALE 1, and to turn it off you should type SCALE 0. If you add this line to the program we have been using so far you will see exactly what the command does

```
15 SCALE 1
```

When you now RUN the program you will see the triangle appear in the top lefthand corner of the screen, and it will be much smaller. This is because the SCALE command makes the computer think that the screen has 1024 pixels across and 1024 pixels down, although it really has 320 across and 200 pixels down (in normal high-resolution mode) or 160 across and 200 down (in multicolour mode). This can be useful for graphs.

PAINT

The PAINT command is used to add a little colour to your pictures. The

command is very simple to use — you just tell the computer which colour source you want to use to choose the PAINT colour from, and the place to start PAINTing from. Delete line 15, type in SCALE 0 (directly) and then type this line

```
30 PAINT 1,160,100,1
```

and when you RUN the program your triangle with be PAINTed white. This is because we have told the computer to PAINT an area in the current text colour, starting at the point 160 pixels across and 100 pixels down. The ',1' at the end tells the computer that you want it to stop PAINTing when it reaches a line which is any colour apart from the background colour. If you leave off the ',1' then the computer will carry on PAINTing until it reaches a boundary which is the same colour as the one which you are PAINTing in.

To select which colour to PAINT in you must change the text colour to the colour you want to PAINT in. To paint your triangle red, for example, you would need to change the text colour, like this

```
25 COLOR 1,3,4
```

Multicolour Graphics

If you have experimented with your own graphics then you may have noticed that you cannot have more than two colours in a single character square (remember the way the screen is divided up into character squares? If not then refer back to the chapter on the CHAR command). If you type this short program

```
10 GRAPHIC 1,1:COLOR 0,1,7:COLOR 4,1,7:COLOR 1,2,7
20 CIRCLE 1,50,50,20:PAINT 1,50,50
30 COLOR 1,5,5
40 CIRCLE 1,70,70,25:PAINT 1,70,70
50 COLOR 1,6,5
60 CIRCLE 1,40,90,15:PAINT 1,40,90
```

you will see circles being drawn on the screen and then PAINTed. You will also see that where the circles overlap whole character squares change colour, because you cannot have more than two colours in one character square, and one of those is the background colour. To overcome this problem we must use one of the multicolour graphics modes. Try changing your program to

```
10 GRAPHIC 3,1:COLOR 0,1,7:COLOR 4,1,7:COLOR 1,2,7
20 CIRCLE 1,50,50,20:PAINT 1,50,50,1
30 COLOR 2,5,5
40 CIRCLE 2,70,70,25:PAINT 2,70,70,1
50 COLOR 3,6,5
60 CIRCLE 1,40,90,15:PAINT 1,40,90
```

When you RUN the program this time the three circles will be drawn and PAINTed and each will be a different colour, with no jagged blocks anywhere! This is because we are using a multicolour mode, in which we can have up to four colours in a single character square (one of these colours being the background colour).

You will notice that for each circle we use a different colour source, each one being defined by the COLOR command. The second circle is drawn in multicolour 1 (which is colour source 2) and the third circle is drawn in multicolour 2 (which is colour source 3). The first circle is still drawn in colour source 1 (the normal text colour source).

You may have noticed, however, that the circles look a little more chunky. This is because when we use a multicolour mode the horizontal resolution halves, so instead of having 320 pixels across the screen we only have 160. So, for highly detailed pictures which do not need a great concentration of colour, it is better to use GRAPHIC 1 or GRAPHIC 2, but where colour is more important it is best to use GRAPHIC 3 or GRAPHIC 4.

One special thing which we can do with colour source 3 when in multicolour mode is to change the colour of everything which was drawn using that colour source. For instance, if you drew a circle using colour source 3 which had been set to white, then a white circle would be drawn. If you then changed the colour of colour source 3 to red, then the circle which you had just drawn would immediately change to red, something which does not happen with any other colour source. Try adding this to your program

```
70 FOR C = 1 TO 8:FOR I = 0 TO 7
80 COLOR 3,C,I:FOR M = 1 TO 500:NEXT M
90 NEXT I,C
```

When you RUN the program now you will see one of the circles (the one drawn in colour source 3) change through all the different colours and luminances possible on your computer.

SSHAPE and GSHAPE

SSHAPE and GSHAPE are two very useful graphics commands, because they allow you to store an area of the screen in a string variable and then put

it back somewhere else. This means that you can move a complex shape, like a spaceship, around the screen smoothly.

The SSHAPE command is the one which stores an area of the screen in a string variable. All you have to do is tell the computer which string you want the shape stored in, and give it the coordinates of the top lefthand corner of the shape and the bottom righthand corner. Here is an example

```
320 SSHAPE SP$,40,50,60,70
```

This line would store the area of memory with the top lefthand corner being 40 pixels across and 50 pixels down, and the bottom righthand corner being 60 pixels across and 70 pixels down, in the string variable SP$.

To put the shape back on the screen you need only tell the computer which variable the shape is stored in, and give it the coordinates of the top lefthand corner of the shape, like this

```
400 GSHAPE SP$,60,70
```

This line would put the shape stored in the string variable SP$ on the screen, with the top lefthand corner of the shape at the point 60 pixels across and 70 pixels down.

If you type in this short program you will see a ball moving slowly across the screen and bouncing off the edges (unfortunately, this means of moving pictures is very slow)

```
10 COLOR 1,6,5:COLOR 0,2,7:COLOR 4,3,4:GRAPHIC 1,1
30 CIRCLE 1,160,100,3:PAINT 1,160,100
40 SSHAPE BALL$,156,96,164,104
50 X = 160:Y = 100:XD = 1:YD = 1
60 GSHAPE BALL$,X-4,Y-4
70 X = X + XD:Y = Y + YD
80 IF X> 315 THEN XD = -1
90 IF X< 4 THEN XD = 1
100 IF Y> 195 THEN YD = -1
110 IF Y< 4 THEN YD = 1
120 GOTO 60
```

Here is an explanation of how the program works

Line 10: Set the DRAW colour to green, the background colour to white and the border colour to red, then select graphics mode 1 and clear the screen

Line 30: DRAW a circle in the current text colour with a radius of 3 pixels with the centre 160 pixels across and 100 pixels down then PAINT it in the current text colour

Line 40: Store the shape with the top lefthand corner at the point 156 pixels across and 96 pixels down and the bottom righthand corner at the point 164 pixels across and 104 pixels down in the string variable BALL$
Line 50: Assign the value 160 to the variable X, the value 100 to the variable Y, and the value 1 to the variables XD and YD
Line 60: Put the shape stored in the string variable BALL$ with the top lefthand corner at the point X−3 pixels across and Y−3 pixels down
Line 70: Add the value of the variable XD to the variable X
Line 80: If the value of X is greater than 315 then assign the value −1 to the variable XD
Line 90: If the value of X is less than 4 THEN assign the value 1 to the variable XD
Line 100: If the value of Y is greater than 195 THEN assign the value −1 to the variable YD
Line 110: If the value of Y is less than 4 THEN assign the value 1 to the variable YD
Line 120: GOTO line 60 and carry on with the program from there

At the moment we are putting the shape onto the screen exactly as it was when we first stored it in the string variable BALL$. However, we can make the computer put the shape back on the screen in a different way. To see how this can be done, alter line 60 to

```
60 GSHAPE BALL$,X-4,Y-4,1
```

When you RUN the program this time you will see the ball move across the screen as before, except this time it is inverted. This means that whereas before the ball was green on a white background, the ball is now white on a green square.

If you now change line 60 to

```
60 GSHAPE BALL$,X-4,Y-4,2
```

and RUN the program you will see a green line with a curved front moving across the screen. This is still the ball, but this time the shape of the ball is being ORed with the background. This means that when the computer puts the shape on the screen and compares every single point on the shape with the point on the screen which it will occupy. If either, or both, of the points are lit then the computer will light up each point when it puts the shape on the screen. This means that you can put one shape over the top of another, so that it looks like one is moving over the other. For instance, if you want to OR the shape ' | ' with the shape '−' you would get the result '+'.

Now, change line 60 to this and see what result you get

```
60 GSHAPE BALL$,X-4,Y-4,3
```

When you RUN the program this time you will see the ball vanish, never to appear again. If, however, you add this line

```
20 COLOR 1,3,4:BOX 1,0,150,319,170,1:COLOR 1,6,5
```

and RUN the program you will see a red band drawn across the lower part of the screen, and when the invisible ball moves across it you will see parts of the red band change to green and a white path will be cut through the band. The reason you cannot see the ball is because it is being ANDed with the background. This means that as the computer puts the shape on the screen it compares every point of the shape with the point on the screen which it will occupy. The computer will only light up a point on the shape if that point is lit up both on the shape AND on the background. For instance, if you ANDed the shape '+' with the shape '|' then the result would be the shape '|' (because that is the only area which is common to both shapes).

Finally, if you change line 60 to

```
60 GSHAPE BALL$,X-4,Y-4,4
```

and RUN the program you will see the ball move across the screen leaving a strange pattern behind it. This is because each point of the shape is being exclusive ORed with the background. XOR is very similar to OR, except that a point will only be lit up if that point is lit up on either the shape or the screen, but not both. This means that if your XOR '|' with '-' you get '+', but if you XOR '|' with '|' you get ' '.

RCLR

RCLR is a function, and it is used to find out what colour is assigned to a colour source. For instance, if we had a line such as this in a program

```
320 Z = RCLR(0)
```

then the variable Z would be given the value of the current background colour. The colour source number (in brackets) is from 0 to 4, just like with the COLOR command.

RLUM

RLUM is another function, similar to RCLR except that is is used to find the current luminance level of a colour source. It is used in the same way as RCLR, e.g. the line

```
1550 C2 = RLUM(4)
```

would assign the current luminance value of the border to the variable C2.

RGR

This is another graphics function, and its use is to find out what the current graphics mode is. RGR is used like this

```
30 F = RGR(0)
```

This line would assign the value of the current graphics mode (0–4) to the variable F. The number in brackets can be any number, as it is a dummy argument (in other words, the computer ignores it).

RDOT

RDOT is yet another graphics function. This one is used to find out information about the pixel cursor, and can be used in these ways

```
100 Z = RDOT(0)
```

which assigns the current X coordinate of the pixel cursor to the variable Z

```
100 PRINT RDOT(1)
```

displays the current Y coordinate of the pixel cursor and

```
100 AA = RDOT(2)
```

assigns the value of the current colour source to the variable AA.

On the next few pages is a listing of the *Artist* program which will allow you to draw pictures on the screen in any graphics mode. It would be a good idea to type it in and read through the explanation of how it works, if only to help you understand the graphics commands.

Artist

This program makes full use of your computer's graphics capabilities in order to allow you to draw pictures on the screen. You can draw circles, triangles, boxes, straight lines, dotted lines, in fact virtually anything you can think of.

When you RUN the program you will first be asked which graphics mode you want to work in. You should reply to this question with a number between 1 and 4. The screen will then clear to a black screen with a black

border, and in the middle of the screen will be a red cross.

The cross is your cursor and you can move it around the screen with the cursor control keys. You can also change the screen colour by pressing S and then the colour that you want the screen to be (from 1 to 8). The screen will then clear (so don't do this if you have a picture that you want to keep) and will become the colour that you have chosen. The border colour can also be changed, simply by pressing E and then the colour of the border that you would like.

The draw colour can also be changed (this changes the colour of the cursor as well). This is done by pressing X and then the colour that you want. The cursor will then change to the colour you have chosen. If you want to change the luminance then you have to press I and then the luminance value that you want. This will not have any effect until you change the colour of something. For instance, if you change the luminance to 7, then change the border colour to red, then the border will be very bright red, but everything else on the screen will remain at the same luminance value.

The simplest of the facilities built into *Artist* is the *Draw* facility. If you press D and then move the cursor around the screen you will see a line being drawn as you go. Pressing D again will cancel this function.

You can also draw circles, or any of the other shapes which you can normally get with the CIRCLE command. To do this you should position the cursor on the screen where you want the centre of your circle and then press C. The screen will clear, but don't worry — your picture isn't lost, and you will be asked questions about the circle, namely the X radius and Y radius, the start angle, the end angle, the angle of rotation, and the number of degrees between each segment of the circle. Pressing RETURN in answer to any of the questions will set that value to zero, or 2 in the case of the 'degrees between segments' question. Once you have entered all the values your circle will be drawn for you.

If you are getting fed up with the slow movement of the cursor then you should press F. This makes the cursor move 5 pixels at a time, instead of 1. Pressing F a second time returns the cursor movement to normal. If you use the *Draw* facility while in fast mode then a dotted line will be drawn.

You can also draw boxes. To do this you must first position the cursor at the point where you want one corner of the box to be (it does not matter which corner) and then press O (for origin). A dot will appear at this point. You should then move the cursor to the opposite corner of your box and press B. Your box will then be drawn.

The *Origin* facility is also used when you want to join two points on the screen. If you move the cursor to the first point and press O, then move the cursor to the second point and press J then a line will be drawn between those two points. Moving the cursor to another place and pressing J again will draw a line from the last point (where you pressed J the first time) to

where the cursor is positioned. You can therefore move all over the screen pressing J and a line will be drawn from where the cursor is to where the cursor was the last time you pressed J.

Of course, you can also paint an area. To do this you should position the cursor anywhere within the area to be painted, make sure that you have chosen the right colour, and press P. The area will then be filled up with colour.

When you are using one of the multicolour modes you will need to be able to select three different colour sources. To change the colour of multicolour 1 you should press the colon and then the colour which you would like (from 1 to 8). To change the colour of multicolour 2 you should press semi-colon, and then the colour which you require. Once you have done this you can choose which colour source you wish to use simply by pressing 1 for normal text colour, 2 for multicolour 1, and 3 for multicolour 2.

Note — If you wish to use this program on a Commodore 16 computer then you will need to miss out all REM statements in the program, as well as any spaces between statements. For example, if you see these two lines

```
80 REM STORE AN EMPTY BLOCK IN BK$
90 SSHAPE BK$,X-3,Y-3,X+3,Y+3
```

then you should leave line 80 out, and change line 90 to

```
90 SSHAPEBK$,X-3,Y-3,X+3,Y+3
```

Unless this is done the program will not work.

Also, if you see something enclosed in square brackets (e.g. IF A$ = "[CRSR RIGHT]") then you should press the key which is indicated in these brackets (in this case the right arrow key).

How the program works

Artist is a long program so I shall not describe in detail of how it works, but I shall give you a run-down of what each section does to help you.

Line 10 sets up the screen, border and text colours, then line 20 assigns values to some of the variables which will be used in the program. Line 30 then clears the screen and asks you which graphics mode you would like. Your answer is checked to ensure that it is a correct one by line 40. If you chose a mode which is unavailable you are asked the question again.

Line 50 works out how many pixels there are across the screen in the mode you have chosen, and then line 60 assigns the number of pixels there are down the screen to the variable Y. Line 70 then selects the correct graphics mode and clears the screen.

Line 90 stores the contents of the screen around the cursor (which is just a blank area because the screen has been cleared) in the variable BK$, and then lines 110–120 draw the cursor on the screen. The cursor is stored in the variable CU$ by line 140, before the screen is cleared again by line 160. The cursor, which has just been stored in CU$, is then put on the screen by line 180.

Lines 190–300 scan the keyboard and jump to various subroutines which change the screen border and colours, draw circles and carry out various other features. The cursor control is carried out by line 320–350.

The *Draw* facility is turned on and off by line 370, and line 390 sets the origin. The *Join* facility is carried out by lines 410–420, while line 440 turns on the *Box* facility. If you have pressed F then line 460 turns the fast move on or off, and line 480 makes the cursor move 5 pixels instead of 1.

It is important to make sure that the cursor will not go off the screen, so lines 500 and 510 check that this will not happen when the cursor is moved. Line 530 rubs out the cursor by putting what was previously in that area back on the screen, before the cursor is moved by line 560. The area of screen which the cursor is about to occupy is stored in BK$ so that the cursor will not wipe out everything beneath it.

Line 590 plots a point on the screen if the *Draw* facility is turned on (or if the *Origin* has just been set, in which case a dot is needed to mark the origin). Line 600 resets the variables Q and P if the dot which has just been plotted has to mark the origin.

If a box needs to be drawn then this is done by line 620. This routine first removes the cursor, then draws the box and stores the area which will be under the cursor (this will be one corner of the box) in BK$. Line 630 sends the program back round to line 170 to put the cursor back on the screen and start checking the keys again.

Lines 650–750 are subroutines which change various colours and luminances. Lines 770–860 ask for the dimensions of the circle and then draw it, and lines 880–890 paint an area in the desired colour. The screen colour is changed by lines 910–930, and the two multicolours are chosen by lines 940–1010. Line 1020 scans the keyboard and assigns numeric value of whichever character has been typed (VAL converts a string to a number) to the variable T.

Artist

```
10 GRAPHIC 0:COLOR 0,1,7:COLOR 4,1,7:COLOR 1,2,7
20 C=2: I=7: S=1: Z=4: V=5
30 SCNCLR: INPUT"WHICH GRAPHICS MODE WOULD YOU
LIKE";GM
40 IF GM< 1 OR GM> 4 THEN 30
```

```
50 IF GM> 2 THEN WD = 160:X = 80:ELSE WD = 320:X = 160
60 Y = 100
70 GRAPHIC GM,1
80 REM STORE AN EMPTY BLOCK IN BK$
90 SSHAPE BK$,X-3,Y-3,X+3,Y+3
100 REM DRAW CURSOR
110 DRAW S,X-3,Y TO X-1,Y:DRAW S,X+1,Y TO X+3,Y
120 DRAW S,X,Y-3 TO X,Y-1:DRAW S,X,Y+1 TO X,Y+3
130 REM STORE CURSOR IN CU$
140 SSHAPE CU$,X-3,Y-3,X+3,Y+3
150 SCNCLR
160 REM PUT CURSOR ON SCREEN
170 GSHAPE CU$,X-3,Y-3,4
180 REM WAIT FOR COMMAND TO BE ENTERED
190 GETKEY A$
200 IF A$ = "1" THEN S = 1
210 IF A$ = "2" THEN S = 2
220 IF A$ = "3" THEN S = 3
230 IF A$ = "X" THEN GOSUB 650
240 IF A$ = "I" THEN GOSUB 690
250 IF A$ = "E" THEN GOSUB 730
260 IF A$ = "C" THEN GOSUB 770
270 IF A$ = "P" THEN GOSUB 880
280 IF A$ = "S" THEN GOSUB 910
290 IF A$ = ":" THEN GOSUB 950
300 IF A$ = ";" THEN GOSUB 990
310 REM CURSOR CONTROL
320 IF A$ = "[CRSR RIGHT]" THEN XD = 1
330 IF A$ = "[CRSR LEFT]" THEN XD = -1
340 IF A$ = "[CRSR UP]" THEN YD = -1
350 IF A$ = "]CRSR DOWN]" THEN YD = 1
360 REM TURN DRAW FACILITY ON AND OFF
370 IF A$ = "D" AND P = 0 THEN P = 1: ELSE IF A$ = "D" THEN
P = 0
380 REM SET ORIGIN
390 IF A$ = "O" THEN OX = X:OY = Y:P = 1:Q = 1
400 REM DRAW LINE FROM ORIGIN TO CURSOR
410 IF A$ = "J" THEN GSHAPE BK$,X-3,Y-3:DRAW S,OX,OY TO
X,Y:OX = X:OY = Y
420 IF A$ = "J" THEN SSHAPE BK$,X-3,Y-3,X+3,Y+3
430 REM TURN BOX FACILITY ON
440 IF A$ = "B" THEN B = 1
450 REM TURN 'FAST MOVE' FACILITY ON AND OFF
460 IF A$ = "F" AND F = 0 THEN F = 1:ELSE IF A$ = "F" THEN F = 0
```

```
470 REM MOVE CURSOR 5 TIMES AS FAR IF 'FAST MOVE' IS
ON
480 IF F=1 THEN XD=XD*5:YD=YD*5
490 REM MAKE SURE THAT CURSOR WILL NOT GO OFF
SCREEN WHEN MOVED
500 IF X+XD<0 OR X+XD>WD THEN XD=0
510 IF Y+YD<0 OR Y+YD>200 THEN YD=0
520 REM RUB OUT CURSOR
530 GSHAPE BK$,X-3,Y-3
540 REM MOVE CURSOR
550 X=X+XD:Y=Y+YD:XD=0:YD=0
560 REM STORE AREA OF SCREEN IN BK$ BEFORE CURSOR
WIPES IT OUT
570 SSHAPE BK$,X-3,Y-3,X+3,Y+3
580 REM DRAW A POINT ON THE SCREEN
590 IF P=1 THEN GSHAPE BK$,X-3,Y-3:DRAW S,X,Y:SSHAPE
BK$,X-3,Y-3,X+3,Y+3
600 IF Q=1 THEN Q=0:P=0
610 REM DRAW A BOX BETWEEN CURSOR AND ORIGIN
620 IF B=1 THEN GSHAPE BK$,X-3,Y-3:BOX 1, OX,OY,X,Y:
SSHAPE BK$,X-3,Y-3,X+3,Y+3:B=0
630 GOTO 170
640 REM CHANGE DRAW COLOUR
650 GOSUB 1020:C=T
660 IFC<1 OR C>8 THEN 650
670 COLOR 1,C,I:RETURN
680 REM CHANGE LUMINANCE
690 GOSUB 1020:I=T
700 IFI<0 OR I>7 THEN 690
710 COLOR 1,C,I:RETURN
720 REM CHANGE BORDER COLOUR
730 GOSUB 1020:B=T
740 IFB<1 OR B>8 THEN 730
450 COLOR 4,B,I:RETURN
760 REM ASK FOR CIRCLE DIMENSIONS
770 GRAPHIC 0,1:COLOR 1,2,7
780 INPUT"X-RADIUS";XR:INPUT"Y-RADIUS";YR
790 INPUT"START ANGLE";SA:INPUT"END ANGLE";EA
800 INPUT"ROTATE ANGLE";RA:INPUT"DEGRESS BETWEEN
SEGMENTS";DB
810 GRAPHIC GM
820 IF DB=0 THEN DB=2
830 REM DRAW CIRCLE
840 COLOR 1,C,I
```

```
850 CIRCLE S,X,Y,XR,YR,SA,EA,RA,DB
860 RETURN
870 REM PAINT AREA
880 GSHAPE BK$,X-3,Y-3
890 PAINT S,X,Y,1: SSHAPE BK$,X-3,Y-3,X+3,Y+3:RETURN
900 REM CHANGE SCREEN COLOUR AND CLEAR SCREEN
910 GOSUB 1020:Q=T
920 IFQ<1 OR Q>8 THEN 910
930 COLOR 0,Q,I:SCNCLR:RETURN
940 REM CHANGE MULTICOLOUR 1
950 GOSUB 1020:Z=T
960 IFZ<1 OR Z>8 THEN 950
970 COLOR 2,Z,I:RETURN
980 REM CHANGE MULTICOLOUR 2
990 GOSUB 1020:V=T
1000 IFV<1 OR V>8 THEN 990
1010 COLOR 3,V,I:RETURN
1020 GETKEY A$:T=VAL(A$):RETURN
```

CHAPTER 10

Functions

DEF FN

You should know by now that a function is a command which takes a variable and does something with it before giving you an answer. Quite often you find yourself saying 'Wouldn't it be useful if the computer had a function that could. . .'. Well, fortunately for us, it is possible to make up your own functions by using the DEFine FuNction command. For instance, if you needed to carry out this calculation several times

((ZZ/5)*32)↑2

then you could define your own function which would carry out this calculation with a line such as this

10 DEF FNA1(ZZ) = ((ZZ/5)*32)↑2

This tells the computer to define a function called FNA1 which will divide a number (the number is represented by ZZ) by 5, multiply the result by 32 and then square the result. To see how it works, try adding these lines and then RUNning the program

```
20 SCNCLR
30 INPUT"TYPE IN ANY NUMBER";N
40 PRINT"((";N;"/5)*32)↑2 = ";
50 PRINT FNA1(N)
60 GOTO 30
```

As you may be able to see, all we have to do to use our new function is to give it a number (which we enclose in brackets), and the computer will carry out a calculation using that number. The computer will substitute the value ZZ in the original definition of the function for the number which we give to the function.

You can define as many functions as you like, but you must give each one a different name. The function name is FN and then any combination of letters and numbers, as long as that combination starts with a letter and does not contain a command (just like with variables). For instance the

function names FN1A and FNPRINTER would not be accepted by the computer.

The functions which you can define can carry out any number of calculations, but can only involve numbers, numeric variables or integer variables. String variables and characters enclosed in quotation marks cannot be used in a function which you have defined.

Function keys

You will almost certainly have noticed the function keys, which are marked from F1 to F7. These keys have commands assigned to them, and to see what these are you should type

```
KEY
```

The computer will then display a list of what each key does.

The KEY command can also be used to alter what each key does — to re-define them. For instance, if you wanted to re-define function key 1 so that it would clear the screen and list the program when it is pressed, you would type

```
KEY 1,"SCNCLR:LIST" + CHR$(13)
```

As you can see, the two commands, SCNCLR and LIST, are enclosed in quotation marks, and are separated by colons. The + CHR$(13) tells the computer that you want the commands to be carried out immediately when the key is pressed (CHR$(13) is the code for the RETURN key, so the computer acts as if the RETURN key had been pressed).

Sometimes you may need to enclose something in quotation marks, and since the commands need to be enclosed in quotation marks this can cause a problem. To overcome this we use the CHR$(34) command, like this

```
KEY 3,"PRINT" + CHR$(34) + "HELLO" + CHR$(34) + CHR$(13)
```

As you can see from the above command, wherever you need the quotation marks to go you must close the quotation marks and add + CHR$(34) + then open the quotation marks and continue with the rest of the command.

You may re-define any of the function keys, including the HELP key, so if you are writing your own program where you need special keys to do special tasks then you can re-define one, or all, of these keys to suit your needs.

Numeric functions

Most of the numeric functions available on your computer have already

been covered, but those which haven't are explained in alphabetical order over the next few pages.

ABS

The ABS command is used to find the ABSolute, or positive, form of a number. Type

```
PRINT ABS(-64)
```

then the computer will display the result 64. A number which is already positive will remain positive, so

```
PRINT ABS(53)
```

will still return the answer 53.

DEC

The DECimal function will convert a hexadecimal (base sixteen) number to decimal, like this

```
PRINT DEC("F2")
```

This will give the result 242.

EXP

The EXP function will raise the mathematical constant e (which is the base of natural logarithms, and has the approximate value 2.71828183) to the power of any number. For instance

```
PRINT EXP(2)
```

will square e, giving the result 7.3890561. This particular function is not used often, so don't be worried if you have never heard of e and logarithms.

LOG

LOG is the inverse of EXP, and is used to find natural logarithms.

```
PRINT LOG(5)
```

will give the result 1.60943791.

SGN

If you ever need to find out if a number is positive or negative, then you will need to use the SGN function. This function will give the result – 1 if the number is negative, 1 if the number is positive, and 0 if the number is zero. Here is an example

```
PRINT SGN( – 54);SGN(0);SGN(992)
```

SQR

This function is used to find the square root of a number. Try

```
PRINT SQR(169)
```

which gives the result 13.

USR

The USR command is used to start a machine code program and to pass a number to that machine code routine. For the full use of this function, see the chapter on the monitor.

Trigonometric functions

Your computer has four trigonometric functions — SIN, COS, TAN and ATN, which, as you might expect, evaluate the sine, cosine and tangent of an angle,and the arctangent of a tangent. If you are not familiar with these functions you need not worry — you will probably not need to use them very often, if at all, and in any case they are explained in detail in a large number of introductory mathematics books. The graphics commands we have already looked at cover most of the applications for which you might otherwise want to use trigonometric functions.

The only important point to remember about the operation of the trigonometric functions is that the argument of the SIN, COS and TAN functions is measured in radians. One radian 57.2957805 degrees (to seven decimal places) so that there are 2*PI radians in one complete rotation. Again, you will find a more complete explanation of why mathematicians use radians rather than degrees in any good mathematics textbook, but all you need to remember at the moment is that to convert an angle from degrees to radians we have to divide by 57.2957805. So if we wanted to know the sine of 40 degrees we would type

```
PRINT SIN(40/57.2957805)
```

Similarly, if we had a tangent and wanted to know the corresponding angle, the ATN function would supply an answer in radians, which we would have to multiply by 57.2957805 to convert to degrees. If, for example, we had a tangent of 0.8391, we would use

```
PRINT ATN(.8391)*57.2957805
```

which gives an answer of approximately 40 degrees.

Other functions

There are a few non-numeric functions which we have not yet covered, which are listed here.

HEX$

The HEX$ function is the opposite of the DEC function. This particular function converts a decimal number to a hexadecimal number which can be stored in a string variable.

```
PRINT HEX$(702)
```

gives the result 2BE.

FRE

The FRE function is a useful one, which tells you how much memory you have left for your program. It is used like this

```
PRINT FRE(0)
```

This will tell you exactly how many bytes of memory you have left. To convert this number to kilobytes of memory you must divide the number of bytes by 1024. The number in brackets after the FRE function can be any number — it doesn't affect the result.

POS

This function will tell you which column the next PRINT statement will start. Like FRE, it has a number in brackets after it, but this number can be any number. Here is an example

Z = POS(0)

SPC

The only main difference between SPC and TAB is that SPC works with the printer, but TAB does not. SPC is used in exactly the same way as TAB and does exactly the same thing, so the command

PRINT SPC(5);"HELLO"

will have the same effect as

PRINT TAB(5);"HELLO"

which is display the message HELLO five spaces from the left of the screen.

CHAPTER 11

Machine code

PEEK and POKE

By now you should have an understanding of BASIC, and soon you will begin to think 'If only I could make the spaceship move faster', or 'Can I make the screen scroll sideways as well as up and down?'. At this point you will be ready to begin exploring the world of machine code programming. Your computer has a facility to make this easier, it's called TEDMON. But before we examine that, we need to learn a little about the way the memory of the computer works, and how you can PEEK into it, and POKE information of your own in.

Your computer has two types of memory, ROM (Read Only Memory) and RAM (Random Access Memory). The working of each of these types of memory is very complex, but I will try to give you a simplified explanation.

Imagine that inside your computer there are thousands of glass boxes, each one with a number on it so that you know which box is which (this number represents a memory address), and that inside each box is a piece of paper with a number between 0 and 255 written on it (this represents the contents of the memory).

Some of the boxes have lids which cannot be opened, and some of them have no lids. The boxes with lids represent the ROM, and the boxes without lids represent the RAM.

Because the boxes are made of glass we can look into all of them to see what number is written on the piece of paper inside. We can also take the piece of paper out of the open boxes (RAM) and put another piece of paper back in with a different number on it. With the closed boxes (ROM), however you can only read the content of the memory, you cannot change it.

There are many different types of ROM. First, there is the Central Processing Unit — this is where, as its name suggests, everything is processed — the 'brains' of the computer if you like. The problem with CPU is that it understands only its own language, not BASIC. For this reason it needs another type of ROM, the interpreter to translate BASIC commands you give it into something it can understand.

You will probably have noticed that when you turn your computer off

any program that is stored in memory is lost. The computer never loses the BASIC language, however, so the RAM must need electricity to retain its contents, but the ROM is a permanent type of memory.

Some of the RAM is used by the computer as work space. Everything on the screen, for example, is stored in RAM. You cannot, therefore, use all the RAM for your program.

The PEEK command allows us to look at the contents of the memory to see what is stored there. You can PEEK into both RAM and ROM.

The POKE command allows us to change the contents of the memory like changing the number on the piece of paper in the glass boxes. You can only POKE to RAM, because the ROM cannot be changed. If you do POKE to a ROM location it will have no effect and no harm will be done.

PEEK and POKE are useful for altering what is on the screen. If you clear the screen and type.

```
POKE 3072,1
```

you will see a letter A appear in the top left-hand corner of the screen. The number 3072 is the memory address of the top left-hand corner of the screen. The number 1 is the ASCII (American Standard Code for Information Interchange) code for the letter A. You can change the number 3072 to any number between 3072 and 4072 you will find that you can make the A appear anywhere on the screen. You can also make different characters appear by changing the 1 to any number between 0 and 255.

If you now type

```
PRINT PEEK(3072)
```

you will see the number 1 appear on the screen (providing the A is still in the corner of the screen). This is because we have told the computer to look into memory location 3072 and display the contents on the screen.

If you try PEEKing to different memory locations from 0 to 65535 you will see the contents of those memory locations (you can try POKEing to these locations as well, but don't be surprised if strange things happen).

Colour memory

You already know how to put characters on the screen by using the POKE command, but it is also possible to change the colour of any character on the screen, and also make it flash, by using POKE. Type in this

```
POKE 3072,1:POKE 2048,128
```

You will see a letter A appear in the top left-hand corner of the screen and it will be flashing.

In order to change the colour of a character on the screen, and to make it flash on and off, you must do three things. First you must decide what colour you want the character to be. The number of the colour will be from 1 to 16, as with the COLOR command. You must then decide what luminance value you require. This value, from 0 to 7, should be multiplied by 16 and added to the number for the colour. If you want the character to flash then you must add 128 to the result. The final value which you come up with should then be POKEd into the relevant memory location.

The colour memory is from 2048 to 3048. The easiest way to work out where you should POKE to is to subtract 1024 from the memory location of the actual character. If a letter A has been POKEd into memory location 3116, which is the third character square across the screen and the second one down, then you would subtract 1024 from 3116, giving 2092, and POKE the colour code that you want into memory location 2092.

Here is a program which POKEs randomly coloured circles into random parts of the screen

```
10 COLOR 4,2,7:COLOR 0,2,7:SCNCLR
20 X = INT(RND(0)*999) + 1
30 C = INT(RND(0)*15) + 1
40 I = INT(RND(0)*6)
50 C = C + I*16
60 POKE X + 2047,C:POKE X + 3071,81
70 GOTO 20
```

The program works like this

Line 20: Choose a random number between 0 and 1, multiply it by 999 and round it down before adding 1 to the result and assigning the result to the variable X

Line 30: Choose a random number between 0 and 1, multiply it by 15 and round it down before adding 1 to the result and assigning the result to the variable C

Line 40: Choose a random number between 0 and 1, multiply it by 6 and round it down before assigning the result to the variable I

Line 50: Multiply the value of the variable I by 16 and add the result to the variable C

Line 60: Store the value of the variable C in the memory location 2047 plus the value of the variable X then store the value 18 in the memory location 3071 plus the value of the variable X

An introduction to TEDMON

Machine code is the language which the CPU inside your computer understands. Programming in machine code allows programs to be executed much faster than in BASIC, because the CPU does not need the interpreter to translate all the commands for it.

This chapter is not intended to be an introduction to machine code programming, as it would be possible to write a whole book on the subject. Instead it is designed to be an introduction to TEDMON, a machine code monitor which is built into your computer and allows you to write machine code programs. TEDMON is ready for you to use. To call it up you simply have to type MONITOR. The computer will then display this on the screen (the numbers may vary)

MONITOR (this is what you typed in)

```
MONITOR

   PC    SR  AC XR YR SP
;FFFF   00  FF  FF  FF F9
```

TEDMON is now operating, and the numbers displayed on the screen, the registers, give you information about the computer. We will come to what these numbers actually tell us later, but for now, let's start by seeing what TEDMON can do.

Displaying the contents of the memory

Try typing

M 8188

You will then see this displayed on the screen, but the characters following the colon on each line will appear reversed

```
>8188 02 A9 5A 4C 94 04 45 4E: .> ZL..EN
>8190 C4 46 4F D2 4E 45 58 D4: DFORNEXT
>8198 44 41 54 C1 49 4E 50 55: DATAINPU
>81A0 54 A3 49 4E 50 55 D4 44: T#INPUTD
>81A8 49 CD 52 45 41 C4 4C 45: IMREADLE
>81B0 D4 47 4F 54 CF 52 55 CE: TGOTORUN
>81B8 49 C6 52 45 53 54 4F 52: IFRESTOR
>81C0 C5 47 4F 53 55 C2 52 45: EGOSUBRE
>81C8 54 55 52 CE 52 45 CD 53: TURNREMS
>81D0 54 4F D0 4F CE 57 41 49: TOPONWAI
>81D8 D4 4C 4F 41 C4 53 41 56: TLOADSAV
>81E0 C5 56 45 52 49 46 D9 44: EVERIFYD
```

The letters on the righthand side of the screen should look familiar to you. The reason for this will become apparent later.

The greater-than sign is at the beginning of each line. This is there to allow you to alter the eight two-digit numbers following it. The full use of this command will explained later.

The four-digit number immediately after the greater-than sign is a memory address in hexadecimal. In decimal notation there are ten different numerals. Hexadecimal, or base 16, however, has sixteen different numerals — 0 to 9 and A to F, where A represents 10, B represents 11, and so on. There are two BASIC functions, HEX$ and DEC convert numbers between decimal and hexadecimal, so if you need to convert between these two bases you should return to BASIC and use these functions to do the conversions for you.

8188 is 33160 in decimal, and is a location in the interpreter ROM. The eight hexadecimal numbers following this are the contents of memory location 8188 and the seven memory locations following it. This means that the content of memory location 8188 is 02, the content of memory location 8189 is A9, and so on.

At the end of each line is a series of reverse-field characters. The eight two-digit hexadecimal numbers are the ASCII codes of these characters. Where a particular character is unprintable a full-stop is displayed instead.

The particular area of memory you have been looking at is the reserved-word table in the interpreter ROM. If you want to continue to examine the ROM you should type 'M' and then RETURN each time you want to see the next section of memory. Alternatively, you could type

```
M 8188 8382
```

in order to have all the reserved-word table displayed. The display will scroll quite quickly, but can be slowed down by holding down the Commodre key.

The first number following the M command is the start location of the memory dump, and the second number is the end location. This means that the above M command tells TEDMON to display the contents of all the memory locations between 8188 and 8382.

It was mentioned earlier that the > command allows you to alter the contents of memory locations. Because the greater-than sign is automatically displayed at the beginning of each line of information you can move the cursor to the number which you want to change and change it. If you try moving the cursor to one of the numbers in the memory dump which has just been changed, then press RETURN, you will see the number you have changed return to its original value. This is because the memory dump on the screen is part of the ROM, and, as you know, you cannot alter the contents of the ROM. If, however, you type

M 3000

you can alter as many of the memory locations as you like, because the memory which is now being displayed is part of the RAM.

You need not display an area of memory before you change its contents. Instead, you can type in a command similar to this

> 324A 3A BD F4

This command tells the computer to store the number 3A in memory location 324A, the number BD in memory location 324B, and the number F4 in the memory location 324C.

You can change between 1 and 8 memory locations at a time using this command, so if you wanted to change the 8 memory locations from 2BC2 onwards you might type

> 2BC2 23 B4 6A DA 9F E2 FC 14

Leaving TEDMON

Leaving TEDMON and returning to BASIC is very easy, all you have to do is type

X

and you will be back in BASIC.

Filling an area of memory

It is also possible to fill an area of memory with a certain value. For instance, if you wanted to fill memory locations 2400 to 2A00 with the value A3, you would type

F 2400 2A00 A3

If you now type

M 2400 2A00

you will see that all these memory locations now contain the value A3.

Hunting for numbers and strings

Another useful feature of TEDMON is the *Hunt* facility. If you type

H 7000 9000 C0

the computer will search through memory locations 7000 to 9000 for all occurrences of the number C0. The results of the search will then be displayed, and should be

80AB 83E4 842D 87AB 89A1 8B10 8BDD 8C12 8EB3 8EE1

Each of these numbers is a memory location which contains the number C0. If you check these memory locations you will see that this is true.

You can also search for all the occurrences of a string of characters. If you type

H 8000 9000 'COMMODORE

then you will get

80CF

If you now type

M 80CF

you will see that the word 'COMMODORE' is, in fact, stored in that area of memory, with the 'C' or 'COMMODORE' stored in memory location 80CF.

If you look at the Hunt command above you will see that we preceded the string of characters that we wanted to search for with a single quote. This tells the computer that we want to search for a string of characters rather than a number.

Transferring blocks of memory

One very useful command is the Transfer command. If you type

T 0C00 0FFF 0BD8

you will see everything on the screen move up one line.

We have actually told the computer to transfer the contents of memory locations 0C00 to 0FFF (which is the screen memory) to memory locations 0BD8 onwards. Since 0BD8 is 40 memory locations before 0C00, this transfer has the effect of scrolling the screen up one line.

As you can see, the Transfer command requires three numbers after it. The first number is the start address of the block of memory you want to transfer, and the second number is the end address of that block. The last

number is the start address into which the block of memory will be transferred.

Writing machine code programs

TEDMON also allows you to write programs in machine code. Try typing in this short machine code program

```
A 2000 LDA #$01
A 2002 STA $0C00
A 2005 LDA #$80
A 2007 STA $0800
A 200A BRK
```

As you type each line of the program the computer will move what you have typed in across the screen, and display some numbers before it. It will also display another A and a number at the beginning of the next line automatically. When you come to the end of the program you should just press the RETURN key. The display will then look like this

```
A 2000    A9 01          LDA > $01
A 2002    8D 00    OC    STA $0C00
A 2005    A9 80          LDA > $80
A 2007    8D 00    08    STA $0800
A 200A    00             BRK
A 200B
```

The letter A at the start of each line is the command *Assemble*, and tells the computer that what you are typing in is a machine code command. The number following the A is the memory address in which the machine code command will be stored. You only have to tell the computer the memory address in which you want to START putting your machine code, and the computer will work out the rest itself.

The numbers which the computer adds before the machine code commands as soon as you press RETURN are the actual machine code equivalents of those commands. Every machine code command has a code number. For example, the command LDA has the code number A9 in the above program. This code number changes according to how the command is being used, as you see later.

Either one or two numbers follow the code number. These numbers follow the machine code command, so in the first line of our program, LDA #$01 has been changed to A9 01, the 01 being the same in both cases.

However, when memory addresses are being used, as they are in the second line, things are slightly different. STA $0C00 becomes 8D 00 0C, which means that the 0C and 00 parts of the number are reversed. This is because the computer likes to have the low byte part of the number, the 00 part before the high byte, 0C part.

Finally we come to the actual commands. I do not intend to cover machine code programming in depth, but here is a brief explanation of what each command in the program is doing.

LDA #$01: LoaDs the Accumulator with the hexadecimal value 01. The accumulator can be thought of as being similar to a variable.
STA $0C00: STores the value of the Accumulator in memory address 0C00 (hexadecimal). This is the address of the top left-hand corner of the screen.
LDA #$80: LoaDs the Accumulator with the hexadecimal value 80 (128 decimal).
STA $0800: STores the Accumulator in memory address 0800. This is the colour memory location for the top left-hand corner of the screen.
BRK: Ends the program and returns control back to TEDMON.

Executing machine code programs

You can carry out this program by typing

```
G 2000
```

As soon as you press the RETURN key a black flashing letter 'A' will appear in the top lefthand corner of the screen.

The *Go* command tells the computer to start executing a machine code program. You have to tell the computer where the start of the program is in memory, so in the above example we told the computer to start carrying out the machine code routine which starts at memory location 2000 (hexadecimal).

Disassembling machine code

As well as assembling a machine code program, i.e. converting it from its form as a series of commands to a form as a series of numbers, TEDMON can also disassemble a machine code program, or convert all the numbers to the commands which they represent. To do this we use the D (for disassemble) command. Try this example

```
D 9000
```

You will then see this appear on the screen

```
. 9000  FO 3C       BEQ   $903E
. 9002  C9 FB       CMP   #$FB
. 9004  D0 03       BNE   $9009
. 9006  4C F7  AE   JMP   $AEF7
. 900B  C9 A3       CMP   #$A3
. 900D  F0 50       BEQ   $905F
. 900F  C9 A6       CMP   #$A6
. 9011  18          CLC
. 9012  F0 4B       BEQ   $905F
. 9014  C9 2C       CMP   #$2C
```

This disassembly is in the format which the computer converts your machine code programs to when it assembles them, as you will see if you compare them, although the commands and numbers will be different. If the area of memory which you have disassembled is in RAM then you can alter the memory contents by moving the cursor over the command which you want to alter, and then altering it. If you type

```
D 2000 200A
```

you will see your machine code program displayed on the screen. Move the cursor to the first line and make this alteration

```
. 2000 A9 01 LDA #$02
```

As soon as you press the RETURN key this line will change to

```
A 2000 A9 02 LDA #$02
```

The full-stop at the beginning of the line will change to a letter A (for assemble), and the number 01 will change to 02. The full-stop at the beginning of the next line will also change to a letter A and the cursor will be over the first digit of the number 2002.

The disassemble command takes the same format as the memory dump command, so if you type

```
D 4000 4020
```

then the computer will disassemble the contents of memory locations 4000 to 4020 (hexadecimal).

Comparing blocks of memory

A slightly less useful command is the *Compare* command. This command will compare one block of memory with another, and tell you where the differences are. If you type

```
C 1000 2000 4000
```

then the screen will fill up with memory locations, many of the memory locations from 1000 to 2000 are different from the memory locations 4000 to 5000.

What we told the computer to do was to compare memory locations 1000 to 2000 with locations 4000 to 5000. We need to give the computer only the start address of the second block of memory, because it works out how much it needs to compare.

The computer will display a list of memory locations which differ, so if the two blocks ofmemory are the same then the computer will not display anything. However, if the blocks of memory are totally different then the screen will rapidly fill up with memory addresses.

Saving machine code programs

If you have written a machine code program then you will probably want to save it on tape or disk. TEDMON will do this for you, all you have to do is tell it the program's name, whether you want it saved on tape or disk, and the start and end memory locations. So, if you wanted to save a program called LEFT SCROLL on tape, and that program was stored in memory locations 3000 to 3040 you would type

```
S "LEFT SCROLL",1,3000,3041
```

You will notice that we have to add one to the end memory address, because the save routine will save all the memory betwen the start location up to, but not including, the end location.

If you wanted to save your program on disk you would have to change the device number (the 1 in the above save command) to an 8, like this

```
S "LEFT SCROLL",8,3000,3041
```

Loading machine code programs

Loading machine code programs is easy. All you have to do is tell TEDMON the name of the program you wish to load, and whether you want to load from tape or disk. So, if you wanted to load a machine-code

program with the name FROG from tape you would type

```
L "FROG",1
```

And to load the same program from disk you would type

```
L "FROG",8
```

The ',1' after the name tells TEDMON that you wish to load from tape, whereas ',8' tells TEDMON to load from disk.

Verifying machine code programs

It is possible to verify a machine code program in exactly the same way as it is possible to verify a BASIC program. To do this you must tell TEDMON the name of the program to be verified, and whether it is stored on tape or disk, in exactly the same way as when you load a machine code program, like this

```
V "COMPOSER",1
```

this tells the computer to compare the program on tape with the name COMPOSER with the machine code program currently in memory. If the two programs are the same then the flashing cursor will return, and if they are different the message VERIFY ERROR will be displayed.

To verify a program on disk with the name COMPOSER you would type

```
V "COMPOSER",8
```

As with their BASIC equivalents, the save, load and verify commands all make the screen go blank while using the tape, and the usual messages are displayed.

The registers

Finally, we move on to the registers. These are the letters and numbers which are displayed when you first enter TEDMON, and give you certain pieces of information about the computer. We will go through these one by one.

The first register is labelled PC, and is called the *Program Counter*. This register always points to the part of a machine code program which the computer is carrying out. For instance, if the computer was carrying out a machine code program from memory location 3A42 onwards then the program counter would contain the memory address 3A42, then 3A43, then 3A44 and so on.

The second register is the *Status Register*. This register contains information about the operations which have just been carried out.

Next in the line of registers is the *Accumulator*. This can be thought of as being similar to a variable, although it is used in a slightly different way from variables. The X and Y registers are similar to the accumulator, although each can do certain things which the others cannot.

The last register is the *Stack Pointer*. The stack is an area of memory in which numbers are stored by the computer, and can be thought of as being similar to a pile of books. You can put more books on the pile, and take books away from the pile. However, you can only put books on the top of the pile, not half way down, and you can only remove books from the top, one at a time. This means that the last book added to the pile is the first to be removed, and in the same way the last number added to the pile is the first to be removed. The stack is 256 bytes long, so can hold 256 bytes. The stack pointer points to the first free memory location in the stack.

You may examine the registers at any time by typing

```
R
```

The registers and their contents will then be displayed. You may, if you wish, alter the contents of the registers. The semi-colon command allows you to do this, and as this command is already displayed at the start of the register contents, all you have to do is move the cursor to the contents of the register you wish to change, and change it. For instance, if the registers had these values

```
 PC     SR AC XR YR SP
;BCB1   00 AF BF 28 F9
```

and you wanted to change the contents of the X register to 2A, you would move the cursor so that it was over the B of BF, like this

```
 PC     SR AC XR YR SP
;BCB1   00 AF BF 28 F9
```

then type 2A and press RETURN. The contents of the X register will then change to 2A.

The SYS command

If you want to execute your machine code program from BASIC then you will need to add an RTS instruction at the end of the routine. For example, if you disassemble your short machine code routine and make this alteration

```
A 200A RTS
```

then exit from TEDMON, and type SYS DEC("2000") then your machine code program will be carried out, and the flashing A will appear in the top left-hand corner of the screen.

The SYS command should be followed by the start address of the machine code program. In the above example we have also included the DEC function, which will convert the number 2000 to decimal before the SYS command carries out the machine code program starting at that memory location.

The USR function

An alternative way to execute a machine code program from BASIC is to use the USR function. This function can be used to pass a number or string of letters to a machine code program, and also to execute a machine code program. However, you must store the start address of the machine code program in memory locations 1281 and 1282. For instance, to execute our machine code program using the USR command we must carry out this procedure

```
PRINT DEC("2000")
8192

READY
PRINT 8192/256
32

READY
POKE 1281,0:POKE 1282,32
READY
X = USR(0)
```

The program will then be executed

What we have actually done is convert the hexadecimal number 2000 into decimal, then divide that number by 256. We then POKEd the low byte into memory location 1281, and the high byte, which is 32, into memory location 1282. Because USR is a function, it must be used in the format.

```
variable = USR(value)
```

The value in brackets is the one which you want to pass to the machine code program. Because we do not need to pass a value to our particular program, it does not matter what value we use. The variable which is used will

have a value stored in it when the machine code routine ends, so do not use a variable which is being used for something else.

You should now have a reasonable understanding of how to use TEDMON. When you feel that you would like to learn machine code programming it would be a good idea if you re-read this chapter, so that you know for certain how to use the many facilities of TEDMON.

CHAPTER 12

Peripherals

Using disk drives

Disk drives are extremely useful because they allow you to save and load your programs much faster than is possible with a cassette recorder. You may use either the C 1541, the C 1542 (the only difference between these is the colour) or the SFS 481 fast disk drive, all of which are made by Commodore.

The C 1541 and C 1542 disk drives should be plugged into the socket marked SERIAL at the rear of your computer, and the other end of the lead should be plugged into either of the two din sockets at the back of the disk drive. The SFS 481 disk drive should be connected to the socket marked USER PORT at the back of your computer.

Once you have your disk drive connected you will need some disks to save your programs on. You will need five and a quarter inch single sided, double density disks, which may be hard or soft sectored. Most computer shops, and even some large stationery shops, sell these disks.

Precautions

There are a few precautions which you must take with disks. The actual disk itself is inside a square plastic envelope which is designed to protect the disk inside. There is a slot in this envelope through which you can see the actual disk. You must not touch the disk through this slot. The actual disk itself is made of a similar material to normal cassette tapes, and, like cassette tapes, disks should not be put near a magnet (which includes your television and the top of your disk drive), neither should they be exposed to bright sunlight. Disks should be at a temperature between 10 and 51 degrees centigrade, or 50 to 125 degrees Fahrenheit, and you should always take care never to bend a disk.

The write-protect tab

Before you can save anything on your disks you must format them. To do this you must first make sure that the notch on the left of the disk is not covered. This tab is similar to the tab on a cassette tape. If this tab is

covered then you cannot save anything on the disk, or do anything to alter the contents of the disk. This tab is called the Write-Protect tab.

Once you have made sure that the write-protect tab is not covered you must put the disk in the disk drive. Open the disk drive door by pushing it inwards. You must then insert the disk into the drive with the write-protect tab on the left and the slot through which you can see the disk facing towards the disk drive. Then close the drive door by pushing it down. If you do this properly then the door will click into place.

Initialising the Disk

Now that you have the disk inserted correctly into the disk drive you will need to initialise the disk. The computer expects the disk to be divided into tracks and sectors — there should be 35 tracks with between 17 and 21 sectors to each track. Each sector can hold 256 bytes. In order to set the disk up in this format we must use the HEADER command. So, make sure that you have a disk in the drive and type in

```
HEADER "DISK1",I01,D0,U8
```

Once you have done this the computer will ask you the question

```
ARE YOU SURE?
```

to which you should reply Y, since you are sure that you want to initialise this disk. The disk drive will then start running and the red light will come on. It takes quite a long time to initialise a disk, so you will have to sit back and wait until the drive stops running.

If the red light on the disk drive continues to flash when the drive has stopped running then the disk was not initialised properly. To find out what went wrong you must type in

```
PRINT DS$
```

The computer will then give you an error message and the light on the disk drive will stop flashing. You should then check that the disk is inserted correctly, and that the write protect tab is removed. Try to initialise the disk two or three times, and if it still fails to initialise try another disk. If this second disk initialises correctly then the first disk will probably be faulty. If, after several attempts with several disks, you still cannot get a disk to initialise properly you should take the disk drive back to where you bought it from and explain the problem.

What you actually told the computer to do when you typed in the HEADER command was to divide the disk up into the correct number of

tracks and sectors,and to give the disk the name DISK 1. You also gave the disk an identity number, 01, which the computer put on every sector of the disk. The D0 told the computer that you wanted to initialise the disk in drive 0 (the first drive), and the U8 is added because 8 is the device number of the disk drive.

The disk DIRECTORY

As well as dividing the disk up into the correct format, the HEADER command also sets up an index, or directory, on the disk. You can find out which programs are stored on a disk simply by typing

```
DIRECTORY
```

Since the disk is empty at the moment you will receive this display

```
0 "DISK1       " 01 2A
664 BLOCKS FREE
```

DISK1 is the name of the disk, and 01 is the identity number which you gave the disk. The 2A at the end is an identity number which the computer gives the disk.

You may also display all the files starting with certain characters. For example, if you wanted to display all the files on a disk which started with the characters PROG then you would type in

```
DIRECTORY"PROG*"
```

As you can see, the word PROG is enclosed in quotation marks and is followed by a star. This star tells the computer that you want to see all files which begin with the characters PROG.

You may slow down the speed at which the directory is displayed by holding down the Commodore key. The display can be stopped completely by pressing CONTROL and S,and restarted by pressing any other key.

Saving a program

You are now ready to save a program on disk. The easiest thing to do is to load one of your programs from tape, and as soon as you have done this type in

```
DSAVE"Program name"
```

Where program name is the name you want to give to your program. This

can be any combination of letters and numbers, as long as it starts with a letter and is no longer than 16 characters. The DSAVE command does the same job as SAVE, except that it saves the program on disk instead of tape,and also the screen remains on while a program is saved to disk.

Checking the program

Once the message READY. you can check to see if your program has been saved correctly. To do this you must type in

VERIFY"program name",8

Once again program name is the name of the program which you have just saved. The ,8 is added to the end of the VERIFY command (which you have already used for checking programs on tape) to tell the computer that you want to compare the program on disk with the name program name with the one currently in memory. The VERIFY command works in exactly the same way with disks as it does with cassette tapes, except that the screen doesn't go blank when you VERIFY a program on disk.

If you wish to you can type DIRECTORY and see that the name of the program you have just saved has been added to the index. The number before the program name is the number of blocks taken up by the program (each block is 256 bytes long, so if you multiply the number of blocks taken up by 256 you can find out exactly how many bytes long your program is).

Loading the program

Now that you know that your program is safely saved on disk you can load it back again. Type NEW to delete your program from the computer's memory, and then type

DLOAD"Program name"

In a surprisingly short time your program will be loaded back ready for you to RUN, LIST or do what you like with it.

You do not have to type the whole program name in order to load it. For instance, if you typed

DLOAD"FROG*"

then the computer would load the first program that it finds on the disk beginning with the characters FROG. Alternatively, if you know that the program which you require is the first one on the disk you could type

DLOAD"*"

Changing a program name

It is very easy, and can often be useful to change the name of a program on disk. All you have to do is use the RENAME command, and tell the computer the old name of the program and the name you want it changed to. For instance, if you had a program on your disk called INVADERS and you wanted to change it to COSMIC you would type in

```
RENAME"INVADERS" TO "COSMIC"
```

You can RENAME anything on the disk using this command, and it only takes a few seconds because the computer has only to update the directory.

Making an extra copy of a program

It is often useful to make an extra copy of a program on disk. This is so you have a backup copy if you accidentally wipe out the original. One way of making such a copy is to use the COPY command. For instance, if you wanted to make a copy of a program called CLOCK you would type in

```
COPY "CLOCK" TO "CLOCK2"
```

You will notice that the name of the second copy is different to that of the original. This is because you cannot have two programs with the same name on one disk. COPYing a program takes a little while to do, and the longer the program the longer the COPYing takes.

If you have more than one disk drive then you can COPY a program from one disk to another. So, if you wanted to COPY CLOCK from drive 0 to drive 1 you would type

```
COPY D0,"CLOCK" TO D1,"CLOCK"
```

In this case we can give the second copy of the program the same name as the original because the two programs are on two separate disks.

The COPY command can also be used to COPY everything from one disk onto another if you have two disk drives. To do this you must type in

```
COPY D0 TO D1
```

Erasing programs from disk

At some time or other you are going to need to take a program off a disk. To do this we use the SCRATCH command. For instance, if you wanted to remove a program called ADVENTURE, you would type in

SCRATCH "ADVENTURE"

The computer would then ask

ARE YOU SURE?

to which you should reply Y or N as appropriate.

Re-saving a program

If you want to save an up-dated version of a program in the place of the original one you must add an @ symbol before the program name in the DSAVE command. For instance, if you have updated your program called FROG and want to save the new version over the top of the old version you would type in

DSAVE"@FROG"

and the computer will save your program over the top of the old FROG.

Tidying up the disk

If you have been using a disk for a long time then you will almost certainly have saved and erased several programs, which means there are bound to be a few gaps. Also, if you have been doing any file handling (more about this later) then there may be improperly closed files cluttering up the disk. To tidy things up, and squeeze all the programs as close together as possible you should use the COLLECT command, like this

COLLECT

That's all there is to it — one command. The disk drive will start running and after a short delay the disk will have been tidied up, leaving you more room for storing your programs in.

Making BACKUPs of disks

The BACKUP command is one which can only be used if you have more than one disk drive, because this command copies the contents of an entire disk onto another disk. For instance, to make a BACKUP of everything that is on a disk in drive 0 onto a disk in drive 1 you should type

BACKUP D0 TO D1

The computer will then proceed to copy everything off the first disk onto the second, wiping out anything that was on the second disk. The disk which you are making your BACKUP onto does not need to be initialised, as the BACKUP command takes care of this.

All the disk commands can be used in the same way if you have one disk drive, or two, three or even four. In order to use any of the commands with any disk drive apart from drive 0 you should add ,D and then the number of the drive you wish to use, to the end of whichever command you are using. For instance, if you want to look at the DIRECTORY of the second disk drive you would type

```
DIRECTORY D1
```

and to load a program from the third disk drive you would type

```
DLOAD"program name",D2
```

Using a printer

A printer is an extremely useful device which allows you to make hard copies, or listings, of your programs on paper whih can be filed in case you lose the original copy of the program.

Most printers for your computer are made by Commodore (be careful of those that are not, some do not print the graphics symbols and some need extra cables and interfaces before they will work). The printer is plugged into the socket marked SERIAL. For details of how to connect up the printer to the mains and where the various leads plug into the printer you should refer to your printer manual.

Before you can send anything out to the printer you must first OPEN a channel through which the computer can send the information to the printer. To do this, we must use a command like this

```
OPEN 1,4
```

This command tells the computer to open a channel to device number 4, the printer, and to give it the file number 1. Whenever you want to send something out to the printer you refer to it by the file number which you have chosen. This file number can be any number between 1 and 255, so you could equally well use the command

```
OPEN 54,4
```

The reason that you have to use a file number is that you can have more than one channel open at one time, and giving each channel that you open a

file number makes it easier to swap between them, as you will see later.

A third number may be added to the end of the OPEN statement. This number is the *secondary address*, and may be a 0 or a 7 when using a printer. Choosing a secondary address of 0 tells the computer that you want to print in upper case, and a secondary address of 7 tells the computer that you want to print in lower case.

Having OPENed a channel you need to decide what you want to send out to the printer. If you want to print only a few messages then you have to use the PRINT # command. We will assume that this is what you want to do. Try this example

```
OPEN 1,4:PRINT # 1,"THIS IS A TEST PRINT"
```

As soon as you press the RETURN key the message THIS IS A TEST PRINT will be printed. If you had already OPENed the channel with the file number 1 then you will receive an error message. If this happens then type CLOSE 1 and start again.

You will notice that we have used the file number which we used to OPEN the channel in the PRINT # command. This is because we always refer to the printer by its file number, and not its device number.

You can print anything on the printer that you can normally print on the screen, but PRINT USING and PRINT TAB do not work in quite the same way. More about those commands in a moment.

When you have finished printing messages you must CLOSE the channel, like this

```
CLOSE 1
```

This command closes the channel which has the file number 1 so that information can no longer be sent to the printer.

The second way you can send information out to the printer is by using the CMD command. This command makes the computer send everything out to the printer which would normally go to the screen. So, if you type a short program and then type

```
OPEN 1,4:CMD 1:LIST
```

The computer will OPEN a file to the printer, giving it the file number 1 and then print the program out on the screen. From now on, anything which the computer normally sends to the screen will be sent to the printer. So, if you type

```
PRINT"THIS IS ANOTHER TEST PRINT"
```

the message THIS IS ANOTHER TEST PRINT will be printed by the printer.

Once the program has been LISTed you have to close the channel again, but before you do this you must tell the computer to stop sending information to the printer and to send it to the screen instead. To do this you must type

```
PRINT # 1
```

This command will tell the computer to start sending everything to the screen again, but the channel to the printer is still open. To close the channel again you must CLOSE the channel again, like this

```
CLOSE 1
```

Tape file handling

It is often quite useful to be able to store data on tape, especially when you are writing a long program and are running short of memory in which to store the program. Fortunately for us, it is possible to save information on tape and to load it back into variables.

Before we can save any information on tape we must first OPEN a channel to the tape, in the same way as we opened a channel to the printer, so type

```
10 SCNCLR
20 OPEN 3,1,2,"FILE1"
```

The OPEN command on line 20 tells the computer to OPEN a channel to the tape (device number 1) and to give it the file number 3. The second number is the device number, which for the tape is 1, and the third number tells the computer what you want to do with the file. This number could be

0 Input
1 Output
2 Output with End of Tape marker

In our OPEN command we have chosen output with End Of Tape marker. This tells the computer that when it has finished storing information on tape it should put a marker on the tape to indicate that there is no more information in that file. If this marker is not added then the computer would continue to search for information, even when it had reached the end of the file.

Now that we have opened the file we need to send something out to it. We

use the PRINT # command to output data to the tape in a similar way to the way we send out data to be printed by the printer. Add these lines to your program

```
30 PRINT #3,"HELLO";CHR$(13);"THERE";CHR$(13);"EVERY
BODY"
40 PRINT #3,1;CHR$(13);2;CHR$(13);3
50 CLOSE 3
```

Line 30 of the program tells the computer to send out the words HELLO, THERE and EVERYBODY to file number 3, which is the file we opened to the tape recorder. Each word is separated by a CHR$(13), which is the equivalent of pressing the RETURN key. This character code is added in order to separate the words on tape. The computer requires this character code as a separator to prevent any confusion of the data when it is loaded back from tape.

Line 40 is similar to line 30, except that the numbers 1, 2 and 3 are being output to the tape instead of words. The numbers are still separated by CHR$(13).

Once we have finished storing our information on tape we must close the channel, which is what line 30 does. The CLOSE command is used here in exactly the same way as it is used to close a channel to the printer.

Now to run the program. Find a blank tape and put it into the cassette recorder, then fast-forward the tape to just past the leader. Now type RUN, and press the PLAY and RECORD buttons on the cassette recorder. The screen will then go blank while the computer stores the information in the program on tape.

Once the information has been recorded, type NEW and then enter this program

```
10 SCNCLR
20 OPEN 3,1,0,"FILE1"
30 INPUT #3,A$,B$,C$
40 INPUT #3,A,B,C
50 CLOSE 3
60 PRINT A$;" ";B$;" ";C$
70 PRINT A;B;C
```

Line 20 of this program opens an input channel to the tape recorder. The first number of the OPEN command is the file number, the second number is the device number and the third number tells the computer that you want to input information from that device. FILE1 is the name of the file.

Lines 30 and 40 read in the information from the tape using the INPUT # statement. This statement tells the computer to read in informa-

tion from a device other than the keyboard, and works in a similar way to the normal INPUT statement. In this case, the computer is being told to read in three strings, which are to be stored in the string variables, A$, B$ and C$, and in line 40 we are telling the computer to read in three numbers, which are to be stored in the variables A, B and C.

We use the CLOSE command to close the channel again, and then lines 60 and 70 display the contents of the variables A$, B$, C$, A, B, and C.

Now rewind the tape and type RUN. The computer will tell you to PRESS PLAY ON TAPE, and as soon as you have done this the screen will go blank while the computer reads in the information from the tape . The screen will then return to normal and the information read in from the tape will be displayed on the screen.

Another means of retrieving information from tape is the GET # command. This is similar to INPUT # except that it only reads in one character at a time. Try typing in this program

```
10 SCNCLR
20 OPEN 3,1,0,"FILE1"
30 FOR N = 1 TO 26
40 GET #3,A$
50 PRINT A$;:NEXT
60 CLOSE 3
```

When you RUN this program you will still be asked to PRESS PLAY ON TAPE, but this time the computer will read in what is on the tape one character at a time, and display it on the screen as it goes along.

The GET # statement in line 40 reads in information from the tape one character at a time, and assigns each character to the variable A$. This is then displayed on the screen.

Because the GET # command reads in one character at a time, it also reads in each CHR$(13), so when this character is displayed on the screen it acts in the same way as if you pressed the RETURN key.

Summary

The OPEN statement is used to open a channel to a device so that information can be sent out to it.

The PRINT # statement is used to send out information from a previously OPENed channel.

Information can be read in from a device using the INPUT # statement. This statement reads in a whole series of characters at once.

An alternative way to read in information from a device is to use the GET # statement which reads in one character at a time.

When you have finished using a device you must CLOSE the channel to that device.

Disk file handling

Although it is useful to be able to store information on tape, it is also slow. Disk drives, however, are much faster, so are ideal for file handling.

Saving information on disk is very similar to saving information on tape. Try typing in this short program.

```
10 SCNCLR
20 OPEN 1,8,2,"DISK FILE,S,W"
30 PRINT #1,"THIS IS A SHORT MESSAGE";CHR$(13);"AND THIS
IS ANOTHER"
40 PRINT #1,1;CHR$(13);2;CHR$(13);3
50 CLOSE 1
```

The OPEN command is used in a similar way to open a channel to the disk drive as it is to open a channel to the cassette recorder. The first number is the file number, the second number is the device number (usually 8 for the disk drive), and the third number is the data channel, which can be any number from 2 to 14. The letters in quotation marks are the name of the file, apart from the ',S,W' part. The ',S' tells the computer that you want to use a sequential file, in other words, a file where all the information is stored in the order that you save it in. ',W' tells the computer that you want to write, or send information out, to the file.

Information is sent out to the disk drive in exactly the same way as it is sent out to the cassette. Lines 30 and 40 send out the information using the PRINT # statement, and once again the separate pieces of information are separated by CHR$(13)

Of course, you must CLOSE the file after you have finished with it, and this is done by line 50.

When you RUN the program the disk drive will start up and the information will be recorded. This does not take very long owing to the speed of the disk drive.

Reading the information back from the disk is just as easy as recording it. Type NEW and then enter this program

```
10 OPEN 1,8,2,"DISK FILE,S,R"
20 INPUT #1,A$,B$,A,B,C
30 CLOSE 1
40 PRINT A$:PRINT B$:PRINT A;B;C
```

Line 10 OPENs a channel to the disk drive in the same way as it did in the

first program, the only difference being that the filename ends in ',S,R', indicating to the computer that you wish to read from a sequential file.

The information is read in by line 20 using the INPUT # statement, in the same way as it is read in from the tape. The file is then CLOSEd by line 30 before line 40 displays the information which has been read in.

As with tape files, you may also use the GET # statement to read in information from disk. Try this program

```
10 OPEN 1,8,2,"DISK FILE,S,R"
20 FOR N = 1 TO 50
30 GET # 1,A$:PRINT A$;
40 NEXT:CLOSE 1
```

The GET # statement works in exactly the same way with disk files as it does with tape files, as you will see when you RUN this program.

Summary

When OPENing a data file to the disk you must specify that you require a sequential file, and whether you wish to read from or write to that file.

The INPUT # statement is used to read in long pieces of information, as with tape files.

Again, like with tape files, the GET # statement is used to read in one character at a time.

The channel which you are using must be CLOSEd when you have finished using it.

APPENDIX A

List of BASIC words

Here is a list of all the commands, statements and functions which you have learnt, together with a brief description of what they do. Wherever a lower-case letter is used, e.g. ABS(a), this indicates that this letter may be replaced by any number. If a section of a command is enclosed in square brackets, e.g. CIRCLE s,xr[,ry,sa,ea,ra,de] this indicates that the section in brackets is optional.

ABS(a) — Converts the number a to its positive, or absolute, form.

AND — Can be used with IF...THEN e.g. IF A = 1 AND B = 1 THEN PRINT "A" will display the letter "A" only if the value of A is 1 AND the value of B is 1. AND is also a Boolean operator, e.g. PRINT 3 AND 2, will display the value 2.

ASC("a") — Returns the ASCII code for a character.

ATN(a) — Returns the arctangent of the angle a.

AUTO[n] — Automatic line numbering. The value n is the increment of the line numbers. Typing AUTO without any increment cancels the automatic line numbering.

BACKUP Dn TO Dm [,ON Uz] — Copies the contents of the disk in drive number n onto the disk in drive number m. You may also specify which drive unit.

BOX cs,x1,y1,x2,y2 — Draws a box on the screen in the colour source cs, with one corner at co-ordinates x1,y1 and the opposite corner at x2,y2.

CHAR cs,x,y,"string" — Displays the characters string in colour source cs with the first letter of string in character square x,y.

CHR$(x) — Returns the character which has the code number x.

CIRCLE cs,x,y,xr[,yr,sa,ea,ra,de] — Draws a circle or ellipse in colour source cs with the centre at co-ordinates x,y, of radius xr. yr is the y-radius and defaults to xr. An arc may be drawn by specifying the start angle sa and the end angle ea. The shape may be rotated by specifying the rotate angle ra, and different shapes may be drawn by giving the number of degrees between each segment de.

CLOSE f — CLOSEs file number f.

CLR — Resets the values of all variables.

CMD f — Causes all information to be sent to file number f instead of to the screen.

COLLECT [Dn, ON Uz] — Compacts all files on a disk, and removes improperly CLOSEd files. The disk drive number and drive unit may be specified.

COLOR cs,cl,lu — Assigns the colour cl at luminance level lu to colour source cs.

CONT — Causes a program to re-start after it has been stopped using the STOP or END statements, of if the RUN/STOP key has been pressed. If an error has occurred or a line has been altered in any way then you cannot CONTinue a program.

COPY — [Dn,]"file 1" TO [Dm,]"file 2"[,ON Uz] — Makes a copy of a program, either on the same disk, or on another disk in another disk drive.

COS(n) — Returns the cosine of the angle n.

DATA data list — Used to store a list of information which can be read back at as it is needed.

DEF("hn") — Converts the hexadecimal number hn to decimal.

DEC FNva — Defines a function with the name FNva (where va is any legal variable name).

DELETE [start line][-last line] — Erases blocks of lines.

DIM variable list — Reserves enough memory for the array variables in the variable list.

DIRECTORY [Dn,Uz,"filename"] — Displays the contents of a disk in disk drive n on disk unit z. If the name "filename" is used then the computer will display all files with that name.

DLOAD"program name" — Loads a program from disk with the name "program name".

DO[UNTIL Boolean argument/WHILE Boolean argument]program-lines[EXIT]LOOP[UNTIL] Boolean argument/WHILE Boolean argument] — Carries out everything between the DO and LOOP statements UNTIL or WHILE a condition is fulfilled. The condition should be a Boolean argument.

DRAW [cs[,x1,y1] TO x2,y2]/[cs[,x1,y1] TO d;a] — draws a line from the point x,y to the point x2,y2 in the colour source cs. Alternatively this

command can be used to DRAW a line d pixels long at an angle of a degrees.

DSAVE"program name" — Saves the program currently in memory on disk under the name "program name".

ELSE statements — Used with the IF...THEN structure. See IF.

END — Tells the computer to stop carrying out the program.

ERR$(en) — Returns the error message which goes with the error number en.

EXIT — Used to exit from a DO...LOOP.

EXP(x) — Returns the value of e raised to the power of x.

FRE(x) — Returns the amount of memory available for BASIC programs in byte. The value x may be any number.

FOR n = st TO en [STEP in]:statements:NEXT n — Carries out all instructions between the FOR and NEXT statements until the value of the variable n reaches the value en. n starts at the value st and is increased by one (or the value of in) until it reaches the value of en.

GET[KEY]var$, [,var$,var$...] — Scans the keyboard and stores the character on whichever key is being pressed in the string variable var$. If the KEY instruction is added (i.e. GETKEY var$) then the computer will stop carrying out the program until a key is pressed. Both the GET and GETKEY statements can be used to scan for several keypresses by adding a list of variables after the statement.

GRAPHIC mode[,clear]/CLR — Used to select a graphics mode. The value mode should be between 0 and 4, and clear should be either 0 (to leave the screen as it is) or 1 (to clear the screen). The GRAPHIC CLR statements allows memory space previously reserved for graphics to be used for a normal program.

GSHAPE var$ [,x,y,z] — Displays the graphics shape stored in the string variable var$ on the screen. If no position is stated then the shape is displayed with the top left-hand corner of the shape at the pixel cursor. The coordinates of the top left hand corner may be given (x and y). The value z is also optional and is the replacement mode.

GOTO line number — Forces the computer to jump to a line number in another part of the program to continue carrying it out from there.

GOSUB line number — Similar to GOTO except this statement sends the computer to a subroutine. When a RETURN statement is reached the

computer returns to the statement immediately after the GOSUB statement.

HEADER"name"[,Iid, Ddrive,ON Uunit] — This command formats a disk, which must be done before a new disk can be used. If the identification number is left off then a fast format is carried out and only the directory is wiped clean (effectively erasing all programs in the process). This can only be done with a disk which has already been formatted.

HELP — If an error has occurred in a BASIC program, then this command will cause the line with the error to be displayed on the screen with the mistake flashing.

HEX$(n) — Converts a decimal number n to a hexadecimal string.

IF Boolean argument THEN statements[:ELSE statements] — The statements following the THEN statement will only be carried out if the condition defined by the Boolean argument is fulfilled. If it is not then the conditions following the ELSE statement are carried out.

INPUT ["**string**";] **var1,var2,var3**... — Displays the string 'string' (if it is added) and then waits for something to be typed in, which will then be assigned to the variable var (this may be a string, numeric, integer or array variable). Several pieces of information may be INPUTted by putting several variables after the INPUT statement.

INPUT # file number,var,var,var... — Similar to INPUT except reads information in from a device.

INSTR(string1,string2[,start] — Returns the position of the string string2 in the string string1. If the value start is given then the search will start at that position.

INT(x) — Rounds the number x down.

JOY(x) — Returns a value depending on the position of joystick x. If the fire button is pressed then 128 is added to the direction value.

KEY[n,command string] — Displays the commands which are assigned to each function key. These keys can also be redefined using this command.

LEFT$(string,n) — Returns the last n characters of the string string.

LEN(string) — Returns the number of characters in the string string.

LET var = value — Can be used to assign a value to a variable, but is not necessary.

LIST [start line-end line] — Displays all or part of a program on the screen.

LOAD"program name"[,device,re-locate] — Loads the program with the name "program name" into memory from tape, unless otherwise specified. If the value re-locate is set tc 1 then the program is loaded back to the memory location from which it was saved. This is normally only used for machine code programs.

LOCATE x,y/d;a — Locates the pixel cursor at the point x pixels across and y pixels down. Alternatively the pixel cursor can be moved by d pixels at an angle a.

LOG(x) — Returns the natural log of the value x.

LOOP — Marks the end of a DO...LOOP. See DO.

MID$(string$,x,n) — Returns n characters from the middle of a string, starting with the xth character.

MONITOR — Enters TEDMON.

NEW — Erases the program currently in memory, as well as all variables.

NEXT [var] — Marks the end of a FOR...NEXT loop. See FOR.

NOT n — This is a Boolean operator and returns the value of not n. Can also be used with IF...THEN (eg IF NOT A = 1 THEN PRINT"YES" will only display the message YES if the value of A is not equal to 1).

ON n GOTO/GOSUB line1, line2, line3... — Depending on the value of the variable n the computer will jump to one of the program lines in the list.

OPEN file number,device[,secondary address,"filename,filetype,mode"] — Opens a channel to a device, giving it a file number.

OR — Used with IF...THEN (eg IF A = 1 OR B = 1 THEN STOP will cause the program to stop if either the value of A is 1 or the value of B is 1 or both).

PAINT [cs,x,y,mode] — Fills an area of the screen with colour. If the start co-ordinates are not stated then filling starts at the current position of the pixel cursor. If the colour source is not stated then the current foreground colour is used.

PEEK(address) — Returns the contents of the memory location address.

POKE address,value — Stores the number value in memory location address.

POS(x) — Returns the horizontal position of the cursor. x may be any value.

PRINT list — Displays the contents of the list on the screen (e.g. PRINT"HELLO" — displays the word HELLO on the screen, PRINT 4 + 5 displays the result of the calculation 4 + 5).

PRINT # file number, list — Similar to PRINT except this statement sends the list out to the file file number, instead of to the screen.

PUDEF"characters" — Used to re-define the symbols displayed with the PRINT USING statement.

RCLR(n) — Returns the colour assigned to colour source n.

RDOT(n) —Returns information about the pixel cursor (n = 0 for x-position of pixel cursor, n = 1 for y-position of pixel cursor, n = 2 for colour source).

READ var,var,var — Reads information from DATA statements into the variables following the READ statement. var can be any type of variable.

RENAME"old" TO "new" [,Ddrive,Uunit] — Changes the name of a file on disk from its old name old to its new name new.

RENUMBER [new start,increment,old start] — Renumbers a program, with the old starting line old start becoming the new starting line new start in increments of increment. If no start line, increment or old start line are stated then the program is RENUMBERed with the first line becoming line 10, and all following lines in steps of ten.

REM message — Used to add comments to a program. The message after the REM statement is ignored.

RESTORE [line number] — Tells the computer to start taking data from the first DATA statement. If a line number is specified then the computer starts taking data from the DATA statement on that line number.

RESUME [line number/NEXT] — After a TRAP statement is carried out, this statement will cause the computer to attempt to carry out the statement which caused the error a second time. RESUME NEXT will cause computer to carry out the next statement after the one which caused the error. If a line number is specified then the computer will start carrying out the program from that line number.

RETURN — This statement tells the computer to go back to the statement after the last GOSUB statement which was executed.

RGR(n) — Returns the current graphics mode. n is a dummy argument and may be any value.

RIGHT$(string,n) — Returns the first n characters of the string string.

RLUM(cs) — Returns the luminance level of colour source cs.

RND(seed) — Returns a value between 0 and 1. The way in which the number is chosen is defined by the seed.

RUN [line number] — Starts executing the program currently in memory. If a line number is specified then the program is started from that line.

SAVE"program name"[,device number,EOT] — Saves the program currently in memory onto a tape, unless another device number is specified. If ,1 is added at the end then an End Of Tape marker is added.

SCALE 1/0 — Turns the scaling facility on or off.

SCNCLR — Clears the screen.

SCRATCH"file name"[,Ddrive,Uunit] — Removes a program from a disk.

SGN(n) — Returns the value 1, −1 or 0 according to whether n is positive, negative or zero.

SIN(a) — Returns the sine of the angle a, which should be given in radians.

SOUND voice,note,length — Plays a tone note on voice voice for a duration of length.

SPC(n) — Similar to TAB, except that this function also works on the printer, whereas TAB does not.

SQR(n) — Returns the square root of the value n.

SSHAPE var$,x1,y1[,x2,y2] — Stores an area of the screen from coordinates x1,y1, to x2, y2. If x2 and y2 are not specified then they are taken to be the position of the pixel cursor.

STEP increment — Used with the FOR statement. See FOR.

STOP — Halts execution of a program, with a BREAK message.

STR$(n) — Converts the value n to a string.

SYS address — Carries out the machine code program starting at memory location address.

TAB(n) — Moves the cursor to the n + 1th character square across the screen.

TAN(a) — Returns the tangent of the angle a which should be in radians.

THEN statements — Part of the IF. . . THEN. . . ELSE structure. See IF.

TO — Used with the FOR statement. See FOR.

TRAP ln — Forces the computer to jump to line number ln when an error occurs.

TROFF — Turns off the trace facility.

TRON — Turns on the trace facility.

UNTIL Boolean argument — Used in DO. . . LOOPs. See DO.

USING string;list — Used with PRINT to display the contents of the list in a format specified by the string.

var = USR(x) — Begins a machine code routine, the start address of which is stored in memory locations 1281 and 1282. The value x is passed to the machine code program in the floating point accumulator. The variable var will contain a number which is passed back to BASIC from machine code.

VAL(string) — Returns the numeric value of the string string.

VERIFY"program name"[,device,relocate flag] — Compares the program on tape (unless a different device is specified) with the program in memory. If the programs are the same then the message OK is displayed, otherwise a VERIFY ERROR occurs.

VOL v — Sets the volume level to the value v.

WAIT address,value 1 [,value 2] — Execution of the program is stopped until the value of memory location address, when exclusive ored with value 2 then ANDed with value 1 is any value other than zero.

WHILE Boolean argument — Used in DO. . . LOOPs. See DO.

APPENDIX B

BASIC Abbreviations

Most of the commands, statements and functions which your computer understands have shortened forms. These usually consist of the first two or three characters of the instruction followed by a shifted character. A list of abbreviations is given below. Where a character is enclosed in square brackets this indicates that this character should be shifted.

ABS	A[B]	AB
ASC	A[S]	A♠
ATN	A[T]	AT
AUTO	A[U]	A↑
BACKUP	B[A]	B♠
BOX	B[O]	B┌
CHAR	CH[A]	CH♠
CHR$	C[H]	C↑
CIRCLE	C[I]	C↖
CLOSE	CL[O]	CL┌
CLR	C[L]	CL
CMD	C[M]	C↖
COLLECT	COL[L]	COLL
COLOR	CO[L]	COL
CONT	C[O]	C┌
COPY	CO[P]	CO┐
COS	NONE	COS
DATA	D[A]	D♠
DEC	NONE	DEC
DEF	D[E]	D¯
DELETE	DE[L]	DEL
DIM	D[I]	D↖
DIRECTORY	DI[R]	DI←
DLOAD	D[L]	DL
DO	NONE	DO
DRAW	D[R]	D←
DSAVE	D[S]	D♠
ELSE	E[L]	EL
END	E[N]	E↑
ERR$	E[R]	E←
EXIT	EX[↖]	EX↖
EXP	E[X]	E♠
FOR	F[O]	F┌
FRE	F[R]	F←

GET	G[E]	G¯
GETKEY	GETK[E]	GETK¯
GET#	NONE	GET#
GOSUB	GO[S]	GO♠
GOTO	G[O]	G┌
GRAPHIC	G[R]	G←
GSHAPE	G[S]	G♠
HEADER	HE[A]	HE♠
HELP	HE[L]	HEL
HEX$	H[E]	H¯
IF	NONE	IF
INPUT	NONE	INPUT
INPUT#	I[N]	I↑
INSTR	IN[S]	I↑
INT	NONE	INT
JOY	J[O]	J┌
KEY	K[E]	K¯
LEFT$	LE[F]	LE←
LEN	NONE	LEN
LET	L[E]	L¯
LIST	L[I]	L↖
LOAD	L[O]	L┌
LOCATE	LO[C]	LO¯
LOG	NONE	LOG
LOOP	LO[O]	LO┌
MID$	M[I]	M↖
MONITOR	M[O]	M┌
NEW	NONE	NEW
NEXT	N[E]	N¯
NOT	N[O]	N┌
ON	NONE	ON
OPEN	O[P]	O┐
PRINT	P[R]	P♠
PEEK	P[E]	P¯

POKE	P[O]	P┌
POS	NONE	POS
PRINT	?	?
PRINT#	P[R]	P─
PUDEF	P[U]	P╱
RCLR	R[C]	R┬
RDOT	R[D]	R¯
READ	R[E]	R¯
REM	NONE	REM
RENAME	RE[N]	RE╱
RENUMBER	REN[U]	REN╱
RESTORE	RE[S]	RE●
RESUME	RES[U]	RES╱
RETURN	RE[T]	RE┤
RGR	R[G]	R┤
RIGHT$	R[I]	R╲
RLUM	R[L]	RL
RND	R[N]	R╱
RUN	R[U]	R╱
SAVE	S[A]	S♠
SCALE	SC[A]	SC♠
SCNCLR	S[C]	S┬
SCRATCH	SC[R]	SC─
SGN	S[G]	S┤
SIN	S[I]	S╲

SOUND	S[O]	S┌
SPC	S[P]	S┐
SQR	S[Q]	S●
SSHAPE	S[S]	S♥
ST	NONE	ST
STEP	ST[E]	ST¯
STOP	S[T]	S┤
STR$	ST[R]	ST─
SYS	S[Y]	S┤
TAB	T[A]	T♠
TAN	NONE	TAN
THEN	T[H]	T┤
TI	NONE	TI
TI$	NONE	TI$
TRAP	T[R]	T─
TROFF	TRO[F]	TRO─
TRON	TR[O]	TR┌
UNTIL	U[N]	U╱
USING	US[I]	US╲
USR	U[S]	U♥
VAL	NONE	VAL
VERIFY	V[E]	V¯
VOL	V[O]	V┌
WAIT	W[A]	W♠
WHILE	W[H]	W┤

APPENDIX C

CHR$ Codes

Below is a list of the CHR$ codes for all the characters available on your computer. As you will probably see, some CHR$ codes have no effect, and others are doubled up.

0	
1	
2	
3	
4	
5	WHITE
6	
7	
8	DISABLES SHIFT-C=
9	ENABLES SHIFT-C=
10	
11	
12	
13	CARRIAGE RETURN
14	LOWER-CASE
15	
16	
17	CURSOR DOWN
18	REVERSE ON
19	CLEAR SCREEN
20	DELETE
21	
22	
23	
24	
25	
26	
27	
28	RED
29	CURSOR RIGHT
30	GREEN

31	BLUE
32	SPACE
33	!
34	"
35	#
36	$
37	%
38	&
39	'
40	(
41	)
42	*
43	+
44	,
45	-
46	.
47	/
48	0
49	1
50	2
51	3
52	4
53	5
54	6
55	7
56	8
57	9
58	:
59	;
60	<
61	=
62	>

63	?	106	▚
64	@	107	▗
65	A	108	▖
66	B	109	▘
67	C	110	▝
68	D	111	▞
69	E	112	▟
70	F	113	●
71	G	114	▁
72	H	115	▌
73	I	116	▐
74	J	117	▂
75	K	118	▔
76	L	119	▒
77	M	120	▕
78	N	121	🭰
79	O	122	🭱
80	P	123	▃
81	Q	124	🭲
82	R	125	🭳
83	S	126	π
84	T	127	◥
85	U	128	
86	V	129	ORANGE
87	W	130	
88	X	131	
89	Y	132	
90	Z	133	
91	[	134	
92	£	135	
93	]	136	
94	↑	137	
95	←	138	
96	─	139	
97	♠	140	
98	│	141	SHIFT-RETURN
99	─	142	UPPER-CASE
100	─	143	
101	─	144	BLACK
102	─	145	CURSOR UP
103	│	146	REVERSE OFF
104	│	147	CLEAR SCREEN
105	▚	148	INSERT

149 BROWN
150 YELLOW GREEN
151 PINK
152 BLUE GREEN
153 LIGHT BLUE
154 DARK BLUE
155 LIGHT GREEN
156 PURPLE
157 CURSOR LEFT
158 YELLOW
159 CYAN
160 SPACE
161 ▌
162 ▬
163 ▔
164 _
165 |
166 ▒
167 |
168 ▒
169 ◤
170 |
171 ├
172 ▪
173 └
174 ┐
175 ▬
176 ┌
177 ┴
178 ┬
179 ┤
180 |
181 ▌
182 ▌
183 ▔
184 ▬
185 ▬
186 ┘
187 ▪
188 ▪
189 ┘
190 ▪
191 ▚
192 ─
193 ♠
194 |
195 ─
196 ▔
197 ▔
198 _
199 |
200 |
201 ╲
202 ╲
203 ╱
204 └
205 ╲
206 ╱
207 ┌
208 ┐
209 ●
210 _
211 ●
212 |
213 ╱
214 ╳
215 ○
216 ♣
217 |
218 ◆
219 +
220 ▒
221 |
222 π
223 ◥
224
225 ▌
226 ▬
227 ▔
228 _
229 |
230 ▒
231 |
232 ▒
233 ◤
234 |

235 ├
236 ▪
237 └
238 ┐
239 ▬
240 ┌
241 ┴
242 ┬
243 ┤
244 │
245 ▌
246 ▐
247 ▔
248 ▀
249 ▄
250 ┘
251 ▪
252 ▪
253 ┘
254 ▪
255 π

APPENDIX D

ASCII Codes

Shown here is a list of the first 128 characters and their ASCII codes. The last 128 are exactly the same as the first, only in reverse field. So, for instance, if you wanted a reverse-field letter 'H' then you would add 128 to the ASCII code for 'H' (which is 8), giving the result 136. This is the ASCII code for a reverse-field letter 'H'.

0	@	20	T	40	(	60	<	80	┐	100	─
1	A	21	U	41	)	61	=	81	●	101	▏
2	B	22	V	42	*	62	>	82	─	102	▒
3	C	23	W	43	+	63	?	83	●	103	▏
4	D	24	X	44	,	64	─	84	▏	104	▒
5	E	25	Y	45	-	65	♠	85	╱	105	◤
6	F	26	Z	46	.	66	▏	86	╳	106	▏
7	G	27	[	47	/	67	─	87	○	107	├
8	H	28	£	48	0	68	─	88	♠	108	▪
9	I	29	]	49	1	69	¯	89	▏	109	└
10	J	30	↑	50	2	70	─	90	♦	110	┐
11	K	31	←	51	3	71	▏	91	+	111	
12	L	32		52	4	72	▏	92	▒	112	┌
13	M	33	!	53	5	73	╲	93	▏	113	┴
14	N	34	"	54	6	74	╲	94	π	114	┬
15	O	35	#	55	7	75	╱	95	◥	115	┤
16	P	36	$	56	8	76	└	96		116	▏
17	Q	37	%	57	9	77	╲	97	▮	117	▏
18	R	38	&	58	:	78	╱	98	▬	118	▮
19	S	39	'	59	;	79	┌	99	¯	119	¯

120

121

122

123

124

125

126

127

APPENDIX E

Glossary

When reading through computer magazines and other computer books you are bound to come across words which you do not understand. To help you work out what all this computer jargon means, here is a short guide

Acoustic coupler — this is a device which may be connected to a computer, into which a telephone handset may be placed. This allows your computer to communicate with another over the telephone.

Address — this is a number which acts as an index to a memory location.

Assembly language — a programming language in which symbolic instructions are used in order to carry out processes by altering the contents of the computer's memory.

BASIC — Beginner's All-purpose Symbolic Instruction Code. This is the language understood by the majority of microcomputers, including your own.

Bit — this is the smallest unit of information which can be handled by a computer, and can either be a one or a zero.

Bug — a mistake in a program which causes it not to work, or to work incorrectly.

Byte — this is a binary number which is made up of eight bits. Because there are 2↑8 (or 256) combinations of eight ones and zeros, a byte can contain any number between 0 and 255.

Cartridge — a small package containing a program or extra RAM which can be plugged into a computer.

Character set — the set of numbers, letters and symbols which a computer may produce on the normal text screen.

Command — an instruction which is normally typed in directly without using line numbers (eg RUN, NEW, LIST etc).

Compiler — converts a program written in a high-level language such as BASIC to machine code.

CP/M — Control Program for Microcomputers. This is a standard disk operating system which is available for many Z80-based computers. This is a standard language found quite often in business computers. Because it is standard, programs written in CP/M can be easily transferred from one computer to another.

CPU — Central Processing Unit. This is the brains of a computer and is a special type of ROM.

Cursor — this is a symbol which indicates where the next character will appear on the screen.

Data — another word for information.

Debug — to remove all mistakes from a program.

Disk — this is a circular piece of magnetic tape which is enclosed in a protective envelope. Programs and other information can be stored on disks and read back again at very high speeds. Large amounts of information can be stored on disks.

DOS — Disk Operating System. This is a program which is contained either in the computer or the disk drive (sometimes it must be loaded into the computer) which controls the operation of disk drives.

EEPROM — Electrically Erasable Programmable Read Only Memory — this is a special type of ROM which can be programmed (using a special programmer). It can be erased by electrical impulses.

EPROM — Erasable Programmable Read Only Memory — this is similar to an EEPROM, but can be erased by direct exposure to ultra-violet light.

Floppy disk — see disk.

Function — an instruction which takes a number and uses it to perform a task, before returning a result.

Hard copy — A printout of information which is produced by a printer onto paper.

Hard disk — this is similar to a normal disk, but the disks are permanently fixed inside the disk drive. These can store much more information than a normal disk drive, and are much more expensive, mainly due to the accuracy involved in keeping the read head in exactly the correct position over the disk.

Hardware — this is the physical computer — the parts that you can reach out and touch.

Hexadecimal (hex) — this is another name for base 16, which is an alternative way of counting. The letters A-F are used in addition to the digits 0-9 (A = 10, B = 11, C = 12 etc).

High resolution — the measure of resolution is the number of pixels on the screen. The more pixels possible, the higher the resolution.

Instruction — a word which tells the computer to perform a given task.

Interface — this is a device which allows a computer to be connected to another device, such as a printer or disk drive.

I/O — stands for Input/Output. This can be used in reference to a device which reads information in and sends it out, or a port which is capable of sending and receiving information.

Kilobyte (K) — 1024 bytes of memory.

Machine code — or machine language — the language which the CPU understands. This language is made up totally of numbers!

Memory map — this is a table showing what each area of a computer's memory is used for.

Modem — another device which allows communication between computers over a telephone line. This device connects directly to the telephone line and must therefore be approved by British Telecom or the relevant telephone company.

Modulator — this is a device, normally fitted inside a computer, which converts the signal sent out by the computer to a form which can be displayed by a television.

Monitor — this can either refer to a device which does the same job as a television but produces a much better quality picture, or to a program which allows memory locations to be examined and altered.

Parallel — this is the way in which the computer inputs or outputs information. A parallel interface allows several bits of information to be handled at a time.

Pascal — a powerful, high-level language used on some business computers.

Peripheral — this is a device which can be connected to a computer.

Pixel — the smallest point which it is possible for a computer to light up on a television screen.

Port — this is a connection through which information can be input or output.

Printout — see hard copy.

Program — this is a set of instructions which are combined in such a way as to carry out a useful task.

PROM — Programmable Read Only Memory. A special type of ROM which can be programmed by a device designed to carry out such a task.

QWERTY — the name given to the standard keyboard layout.

RAM — Random Access Memory. This is a type of memory whose contents are not permanent, and can therefore be altered. This form of memory can only retain its contents while power is supplied to it.

Register — a part of the CPU which acts as a pointer to a specific block of memory.

ROM — Read Only Memory. This type of memory is permanent and therefore its contents cannot be altered. It is not necessary to supply power to this form of memory in order for it to retain its contents.

Routine — a part of a program designed to fulfill a specific task.

RS232 — a standard form of interface which handles information in a serial form.

Serial — this is a means of inputting or outputting information. The information is handled one bit at a time.

Software — this is a program, and is always stored in hardware of one kind or another.

Source code — this is a program written in a complex high-level language such as BASIC or PASCAL, which needs to be compiled to machine code.

Statement — an instruction which is contained within a program.

String — a series of characters.

Subroutine — see routine.

Toolkit — this is a program which adds to the commands available on a computer.

Utility — similar to a toolkit.

Variable — a value which can be changed. This value is usually represented by a character or series of characters.

VDU — Visual Display Unit. This can be either a television or a monitor.

Z80 — this is a very popular CPU which is at the heart of computers such as the ZX Spectrum, Sharp and Video Genie computers.

6502 — this is another type of CPU, and is the one upon which the CPU in your computer has been based.

APPENDIX F

Some Programs

To finish off this book, here are a few short programs for you to try out

3-D Plot

```
10 GRAPHIC 3,1:COLOR 3,2,7
20 A = 80:B = A*A:C = 100:D = 100
30 FOR X = 0 TO A
40 S = X*X
50 P = SQR(B - S)
60 I = - P
70 R = SQR(S + I*I)/A
80 Q = (R - 1)*SIN(24*R)
90 Y = I/3 + Q*D
100 IF I = - P THEN M = Y:GOTO 130
110 IF Y> M THEN M = Y:GOTO 140
120 IF Y> = N THEN GOTO 170
130 N = Y
140 Y = C + Y
150 DRAW 3,A + X,Y
160 DRAW 3,A - X,Y
170 I = I + 4
180 IF I< P THEN 70
190 NEXT X
200 FOR C = 2 TO 15: FOR I = 0 TO 7:COLOR 3,C,I
210 FOR M = 0 TO 500:NEXT M,I,C
220 GOTO 200
```

Alarm Clock

When you RUN Alarm Clock you will be asked what time you want the alarm to go off. This program works as a twenty-four hour clock, so the alarm time must be entered in the format HHMMSS, where HH is the hours, MM is the minutes and SS is the seconds. Setting the actual clock is

done by holding down the H key to move the hour hand, the M key to move the minute hand, and the S key to move the second hand.

```
10 SCNCLR:PRINT"WHAT TIME DO YOU WANT THE ALARM TO
GO OFF (HHMMSS)";
20 INPUT AL$:IF LEN (AL$)< > 6 THEN RUN
30 VOL 5:V = 5
40 Z = 350
50 GRAPHIC 1,1
60 COLOR 0,3,3:COLOR 4,3,3:COLOR 1,8,6
70 SCNCLR
80 CIRCLE 1,160,100,75
90 FOR N = 0 TO 360 STEP 6
100 DRAW 0,160,100 TO 72;N
110 DRAW 1 TO 3;N
120 NEXT
130 FOR N = 0 TO 360 STEP 30
140 DRAW 0,160,100 TO 65;N
150 DRAW 1 TO 10;N
160 NEXT
170 X = 344:GOSUB 430
180 Z = 344:GOSUB 370
190 TI$ = "000000"
200 IF Z> = 704 THEN Z = 344:GOSUB 370:GOSUB 430
210 FOR A = 0 TO 354 STEP 6
220 SOUND 2,1000,1
230 COLOR 1,2,7
240 DRAW 1,160,100 TO 56;A
250 GOSUB 400:GOSUB 450
260 GET A$
270 IF A$ = "C" THEN GOSUB 470
280 IF A$ = "A" THEN AL = 0
290 IF TI$ = AL$ THEN AL = 1
300 IF AL = 1 THEN FOR M = 1 TO 9:SOUND 1,700,2:SOUND
1,800,2:NEXT
310 IF AL = 1 THEN 330
320 FOR M = 1 TO 310:NEXT M
330 DRAW 0,160,100 TO 56;A
340 NEXT A
350 GOSUB 370
360 GOTO 200
370 DRAW 0,160,100 TO 40;Z TO 18;Z + 34 TO 18;Z – 194 TO 160,100
380 Z = Z + 6
390 E = E + 1
```

```
400 DRAW 1,160,100 TO 40;Z TO 18;Z + 34 TO 18;Z – 194 TO 160,100
410 IF E = 12 THEN E = 0:GOSUB 430
420 RETURN
430 DRAW 0,160,100 TO 30;X TO 14;X + 33 TO 14;X – 192 TO 160,100
440 X = X + 6
450 DRAW 1,160,100 TO 30;X TO 14;X + 33 TO 14;X – 192 TO 160,100
460 RETURN
470 GETKEY A$
480 IF A$ = "H" THEN GOSUB 430:Z$ = STR$(VAL(TI$) + 100000):
Z$ = RIGHT$(Z$,LEN(Z$) – 1)
490 IF A$ = "M" THEN GOSUB 370:Z$ = STR$(VAL(TI$) + 100):Z$ =
RIGHT$(Z$,LEN(Z$) – 1)
500 IF A$ = "S" THEN DRAW 0,160,100 TO 56;A:Z$ = STR$(VAL
(TI$) + 1)
510 DRAW 0,160,100 TO 57;A
520 IF A$ = "S" THEN Z$ = RIGHT$(Z$,LEN(Z$) – 1):A = A + 6
530 DRAW 1,160,100 TO 57;A:GOSUB 450:GOSUB 400
540 IF Z$ = "" THEN 570
550 IF LEN(Z$)< 6 THEN Z$ = RIGHT$("000000",6 – LEN(Z$)) + Z$:
TI$ = Z$:Z$ = ""
560 IF A$ = "T" AND V = 0 THEN VOL 5:V = 5:ELSE IF A$ = "T" THEN
V = 0:VOL 0
570 IF A$ = "C" THEN DRAW 0,160,100 TO 57;A:RETURN
580 GOTO 470
```

Shapes

```
10 GRAPHIC 2,1
20 INPUT"HOW MANY SIDES";A
30 IF A< 2 OR A> 100 THEN PRINT"DON'T BE
RIDICULOUS": GOTO 20
35 SCNCLR
40 CIRCLE 1,160,80,40,33,,,,360/A
50 GOTO 20
```

INDEX

Printed in England by Commercial Colour Press, London E7.

This book has been written with the complete beginner in mind. It is designed to be a combined manual and beginner's course on the Commodore C16 and Plus 4 computers.

The author takes great care not to assume any previous knowledge on the part of the reader. Commands are introduced in such a way that you start programming almost immediately, and their use is illustrated with helpful example programs. As your knowledge of programming increases, the more complicated commands are introduced. By the end of the book you should be proficient in the more sophisticated programming techniques such as disk file handling and high resolution graphics.

This book follows the well-tried methods of teaching BASIC that have made Brian Lloyd's previous book, the Dragon Trainer, such a success.

Brian Lloyd is the author of the Dragon Trainer. He runs his own computer club.

GB £ NET +005.95
ISBN 0-946408-64-5
00595
9 780946 408641
£5.95 net

SUNSHINE
ISBN 0 946408 64 5