BBC Microcomputer System User Guide

WARNING: THE COMPUTER MUST BE EARTHED

Important: The wires in the mains lead to the computer are coloured in accordance with the following code:

Green and yellow   **Earth**
Blue   **Neutral**
Brown   **Live**

As the colours of the wires may not correspond with the coloured markings identifying the terminals in your plug, proceed as follows:

The wire which is coloured green and yellow must be connected to the terminal in the plug which is marked by the letter E, or by the safety earth symbol ⏚ or coloured green, or green and yellow.

The wire which is coloured blue must be connected to the terminal which is marked with the letter N, or coloured black.

The wire which is coloured brown must be connected to the terminal which is marked with the letter L, or coloured red.

If the socket outlet available is not suitable for the plug supplied, the plug should be cut off and the appropriate plug fitted and wired as previously noted. The moulded plug which was cut off should be disposed of as it could be a potential shock hazard if it were to be plugged in with the cut off end of the mains cord exposed. The moulded plug must be used with the fuse and fuse carrier firmly in place. The fuse carrier is of the same basic colour* as the coloured insert in the base of the plug. Different manufacturers' plugs and fuse carriers are not interchangeable. In the event of loss of the fuse carrier, the moulded plug MUST NOT be used. Either replace the moulded plug with another conventional plug wired as previously described, or obtain a replacement fuse carrier from an authorised BBC Microcomputer dealer. In the event of the fuse blowing it should be replaced, after clearing any faults, with a 3 amp fuse that is ASTA approved to BS1362.

*Not necessarily the same shade of that colour.

Exposure

Like all electronic equipment, the BBC Microcomputer should not be exposed to direct sunlight or moisture for long periods.

Econet and The Tube are trademarks of Acorn Computers Limited

The product described in this manual and products for use with it are subject to continuous development and improvement. All information of a technical nature and particulars of the product and its use (including the information and particulars in this manual) are given by Acorn Computers in good faith. However, it is acknowledged that there may be errors or omissions in this manual. A list of details of any amendments or revisions to this manual can be obtained upon request from Acorn Computers Technical Enquiries. Acorn Computers welcome comments and suggestions relating to the product and this manual.

All correspondence should be addressed to:

Technical Enquiries
Acorn Computers Limited
Newmarket Road
Cambridge CB5 8PD

All maintenance and service on the product must be carried out by Acorn Computers' authorised dealers. Acorn Computers can accept no liability whatsoever for any loss or damage whatsoever caused by service or maintenance by unauthorised personnel. This manual is

Within this publication the term 'BBC' is used as an abbreviation for 'British Broadcasting Corporation'.

This book is part of the BBC Computer Literacy Project, prepared in consultation with the BBC Continuing Education Advisory Council.

The editor of the project is David Allen

Note: If this manual is to be used in conjunction with a BBC Microcomputer which is fitted with an Operating System with version number lower than 2.00, then the following points should be borne in mind:

– Chapter 42 and all other references to the 'shadow screen' should be ignored.
– Chapter 49 should be ignored (unless the machine with which the manual is to be used is fitted with BASIC II).

In all other respects this manual is functionally compatible with earlier versions of the BBC Microcomputer.

Contents

Giving the computer instructions – Part 1

Structure in BASIC

Giving the computer instructions – Part 2

Reference section

Introduction

Equipment required

Before you start using your computer check that you have received the following in addition to this User Guide:

– A BBC Microcomputer.
– A guarantee registration card.
– An aerial lead about two metres long which connects the computer to your television.
– The Welcome Package – containing a cassette and an introductory booklet.

If you are short of any of these items then write immediately to your supplier quoting the number given to you when you placed your order. The number also appears on the dispatch label.

You will also require a lead to connect your computer to an ordinary cassette tape recorder. If you ordered the appropriate lead when you placed your order, check that it has arrived. If you didn't, take your cassette recorder, the computer and this book to a dealer and ask if he can supply a lead or make one up for you. In many cases a standard audio lead will be suitable. The most common, useful type is a 5-pin DIN to 5-pin DIN (see below). Alternatively, order the appropriate lead from the supplier of your BBC Microcomputer. Unfortunately, as there are a large number of different kinds of connections, it has not been possible to supply a lead to fit every machine.

Text conventions used in this manual

You will notice that the style of printing used to present the text in this manual varies. This is to help you tell the difference between explanatory text, words which appear on your monitor screen (including BASIC keywords) and certain keys on the computer keyboard.

– Ordinary text appears like this, or *like this* for emphasis.
– Text displayed on the screen (including BASIC keywords) appears `like this`.
– Words like **RETURN** mean that you should press the key marked RETURN rather than type the letters R E T U R N.

What this User Guide can and can't do

The BBC Microcomputer is a very versatile machine. On its own, connected to your television set, the computer can respond to programs which you yourself type in, to produce numbers, words, lines and movement on the screen and sound. Connect a suitable cassette tape recorder and you can then save your own programs for future use or run programs which have been written by other people. The WELCOME cassette which comes with the computer contains a number of programs specially written for the machine. Other programs are available in large numbers, including programs linked to hobbies and games, and programs for use in the home, in business and in education. Languages other than BASIC (such as LISP, FORTH, BCPL and PASCAL) are available. These languages are stored in an integrated circuit which has to be plugged into your BBC Microcomputer. This must be done by your dealer.

The early chapters of this book will show you how to load and save programs from cassette, how to write simple programs and how to create certain graphics effects on the screen. There are also some complete programs to type in yourself. However, this is not a step-by-step course in BASIC programming.

Most of what follows in the later chapters forms a reference guide on how to use the various commands and keywords of the BBC BASIC language. If you are an absolute beginner then much of this will not be very easy to understand. However, as you get more experience of programming, this material will prove invaluable.

1 Getting started

To get your computer working you will need a television set for a screen. Most people at home will use their ordinary colour or black and white television to show the pictures that the BBC Microcomputer produces. You will also need a cassette recorder if you wish to save and load programs.

If you have a high quality monitor (for example in a school) then it can be connected directly to one of the sockets at the back of the computer. To connect the monitor to the computer you will need a special monitor lead.

Assuming that you want to use your normal television set, then you can connect it to the computer using the aerial lead that is supplied with the computer. One of the plugs on this lead has a long central prong which fits into the socket on the back of the computer marked UHF out. The other end of the lead goes into the back of your television set in place of the normal aerial lead (see figure 1). Don't worry about the cassette recorder for the moment.

Next, plug your computer into the mains and switch it on (the On/Off switch is at the back). It should make a short 'beep' and the red light marked caps lock should come on. Turn the television on too and let it warm up for a moment.

Probably all you will see on the TV screen at this stage is a 'snow storm'. You will have to tune the TV so that it can receive the transmissions from the BBC Microcomputer. When your television is tuned correctly words will appear on the screen.

Your television probably has some push-buttons which can be used to select different channels. Often button number 1 is tuned to BBC 1, button number 2 to BBC2, button number 3 to ITV and so on. It is best to tune a spare channel for the computer, for example channel 8. You can then use this for the computer without interfering with the tuning of the normal channels.

Different televisions tune channels in different ways. For some of them, you turn the same knob that you use to select the channel. For others, there are separate controls. In either case, you should depress a spare channel button and then adjust it, or the associated control, until you get a good picture on the screen. A message similar to

```
Acorn OS 64K

BASIC

>
```

4
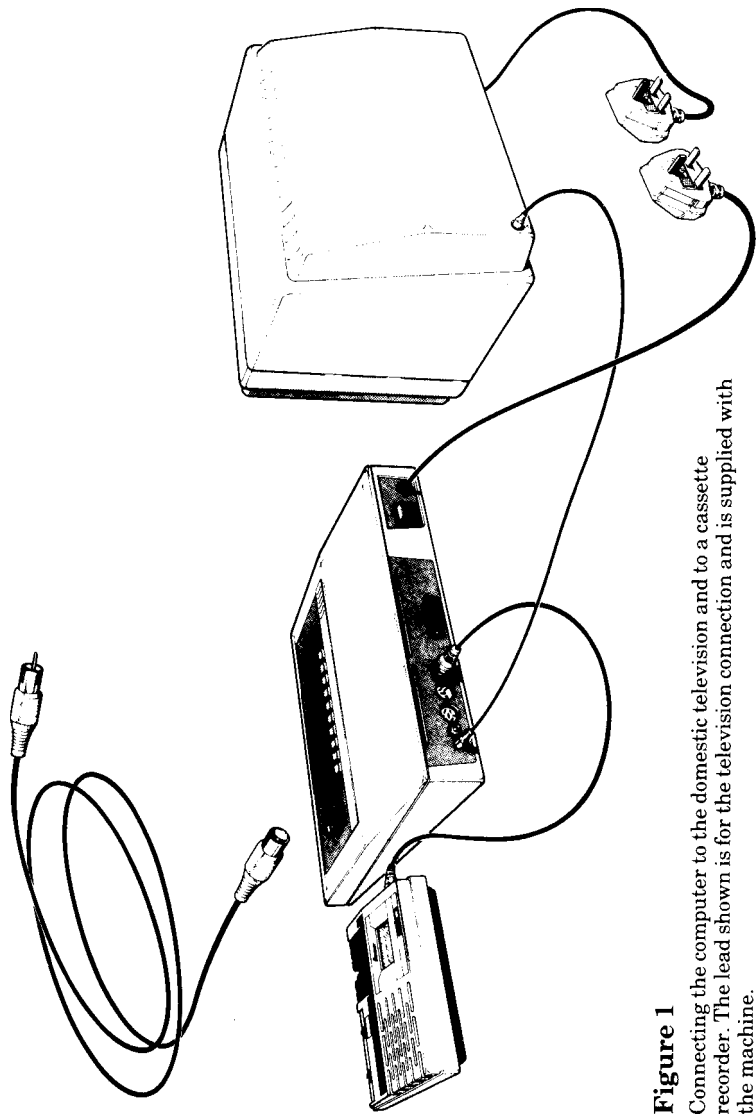
Figure 1

Connecting the computer to the domestic television and to a cassette recorder. The lead shown is for the television connection and is supplied with the machine.

should appear, which should be clear and sharp. Many types of tuning control indicate the channel number that you are tuning to. The BBC Microcomputer transmits on channel 36. It will not be too difficult to find the right channel but you will have to tune the TV carefully to get a really clear picture.

When you have a clear picture, do by all means press every button in sight on the computer – you can't do it any harm at all. Usually it just keeps on saying

```
Mistake
>
```

whenever you press the large key marked RETURN. **Mistake** just means that the computer does not understand your commands. Its fault – not yours!

You will see that if you hold any key down for more than a short time the character on the key appears on the screen, then there is a short pause, then the character repeats until you take your finger off again. On the whole, when pressing keys on the keyboard you should press them briefly – unless you want this repetition.

Experimenting

Now you are ready to experiment. You might like to try some of the following to see what the computer can do, but first be sure to press the key marked BREAK. This will clear the screen and get the computer ready for you.

Type in the following exactly as shown:

```
MODE 5
```

and then press the **RETURN** key. As you will see the command **MODE 5** clears the screen and just leaves the **>** mark on the screen. **>** is known as the 'prompt' and it means that the computer is ready for your next command.

Pressing the **RETURN** key tells the computer that you have finished the line you are typing and that you want it to obey your command. Before you press the **RETURN** key you can correct errors by pressing the key marked **DELETE**.

If the computer says **Mistake** then press the **BREAK** key and try again, starting with **MODE 5**.

Then type in each of the following lines – but don't forget to press the **RETURN** key at the end of every line. Don't worry if you make a mistake – it really doesn't matter!

```
DRAW 1000,100
```

```
DRAW 0,750
```

```
GCOL  0,1
```

```
PLOT 85,0,0
```

If the computer says **No such variable** then you are probably pressing the letter O instead of the number 0.

```
PLOT 86,1000,750
```

```
VDU 19,1,4,0,0,0
```

```
VDU 19,3,2,0,0,0
```

```
VDU 19,0,1,0,0,0
```

```
DRAW 200,0
```

```
DRAW 0,200
```

As you can see, the **DRAW** command is used to draw lines while **PLOT 85** and **PLOT 86** are used to plot and fill in triangles on the screen. When using the graphics the points on the screen are numbered from 0 to 1279 (left to right) and from 0 to 1023 (bottom to top). They are like positions on a piece of graph paper.

Words can also be plotted in colours, as you will have seen. Clear the screen by typing **MODE 5** and then type the following:

COLOUR 1	This selects a red foreground.
COLOUR 2	This selects a yellow foreground.
COLOUR 3	This selects a white foreground.
COLOUR 129	This selects a red background.
COLOUR 0	This selects a black foreground.
COLOUR 130	This selects a yellow background.

The computer can create sound as well. Try typing this in:

```
SOUND 1,-15,100,200
```

and then press **RETURN**.

That gives a simple, crude sound. It is also possible to alter the quality of the sound. Try this:

```
ENVELOPE 2,3,2,-4,4,50,50,50,127,0,0,0,126,0
```

(This should be typed in as one line even though it may spill over to the next line on the screen just as it has on this page. The computer will treat it as being 'one line' when you press **RETURN**.) Now carry on with:

```
SOUND 1,2,1,10
SOUND 2,2,100,1
SOUND 3,2,200,1
```

You will have to press **ESCAPE** to stop the sound.

Here's another one:

```
ENVELOPE 1,1,-26,-36,-45,255,255,255,127,0,0,0,126,0
SOUND 1,1,1,1
```

There is a whole chapter on sound later on.

Connecting up the cassette recorder

Now get a cassette recorder connected so that you can load the demonstration programs into the computer from the cassette tape supplied in the WELCOME pack. For the moment just follow the instructions – we can sort out the 'whys and wherefores' later.

You have to do two things before you can load the programs from the WELCOME tape: first get the right lead to connect your cassette recorder to the computer and secondly set the volume control on the cassette recorder to the correct position.

Leads

There are a number of different kinds of leads (figure 2). The connection to the computer is through a 7-pin DIN connector; a lead has not been supplied with the machine because there are so many connections to the many different cassette recorders in use. In many cases a standard 5-pin DIN to 5-pin DIN lead will be suitable, provided you do not want to use the motor control. If you want full motor control, take your cassette recorder to your nearest BBC Microcomputer dealer who will be able to supply a lead or make one up for you. Alternatively, take your cassette recorder and this book to a local hi-fi dealer.

Note: Although you may find the ideal cassette lead difficult to buy locally, many cassette recorders do have a standard 5-pin DIN socket and a standard 5-pin DIN to 5-pin DIN hi-fi lead will work with the BBC Microcomputer in many cases.

Volume

Having got the cassette recorder connected to the computer the only remaining thing to do is to set the playback volume on the cassette recorder to the correct level.

With the BBC Microcomputer the cassette volume control setting is not critical. However, a special procedure for setting the volume control correctly is incorporated into the first program on the tape.

Running the WELCOME programs

Note: If your machine is fitted with a Disc, Econet, Teletext or IEEE interface and you wish to use a cassette, you must first select the Cassette Filing System by typing

`*TAPE` **RETURN**

This command should also be typed in (if your machine has one or more of the above interfaces) directly after use of **BREAK** or **CTRL BREAK**. In some cases, very long cassette programs may not run because of the small amount of extra memory used by the Disc and Net filing systems. To overcome this, follow the

`*TAPE` **RETURN** command by:

`PAGE = &E00` **RETURN**
`NEW` **RETURN**

(See chapter 33 for explanations of `PAGE` and `NEW` .) If at any time you wish to return to the Disc or Econet filing systems, press **BREAK** or **CTRL BREAK**, or type:

`*DISC` **RETURN**

or

`*NET` **RETURN**

Bearing in mind the above note, the WELCOME programs can be run by typing in

`CHAIN "WELCOME"`

7 pin DIN to 5 pin DIN
and 2.5mm jack

7 pin DIN to
7 pin DIN

7 pin DIN to two 3.5mm
jacks and one 2.5mm jack
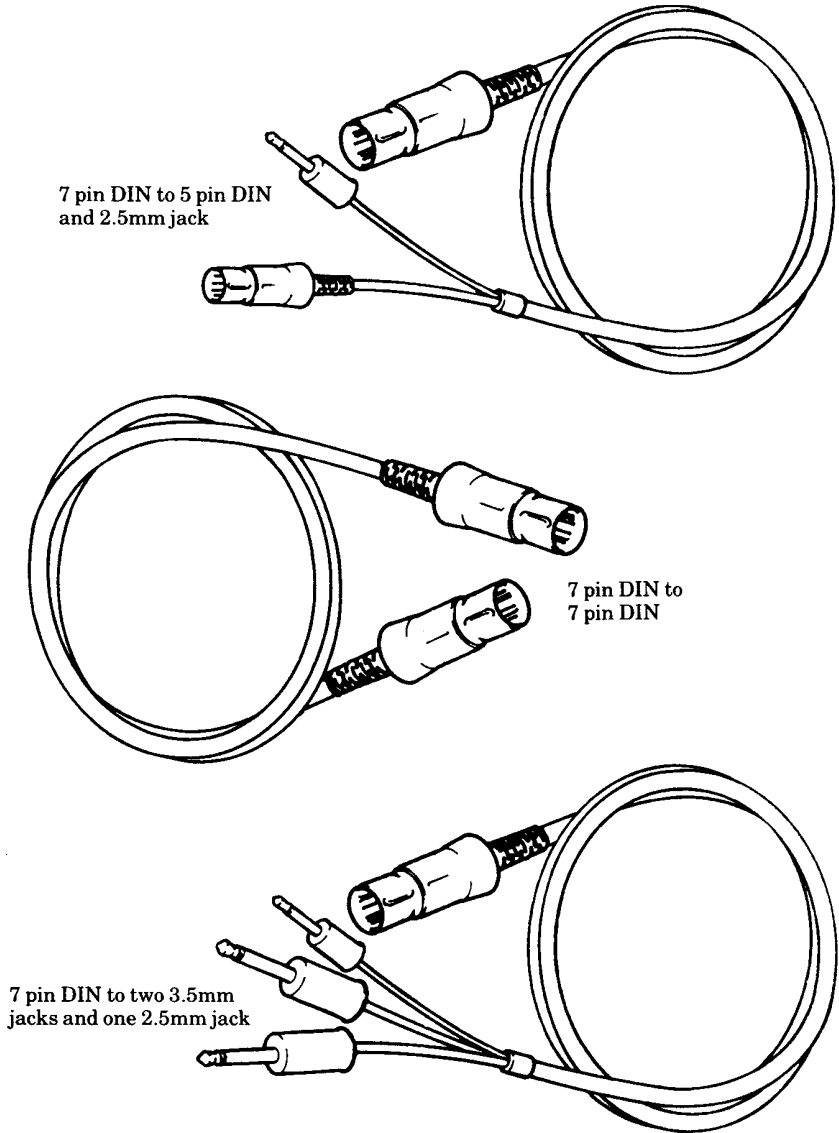
Figure 2 A range of possible cassette leads

You will need to select a lead with a 7-pin or 5-pin DIN lead at one end. This plugs into the computer. The other end of the lead must have suitable plugs for your particular recorder. Note: a standard 5-pin DIN lead will work with many recorders but will not enable you to make use of the computer's ability to start and stop the cassette recorder automatically.

and then press the **RETURN** key. Next insert the WELCOME cassette into your recorder. If your cassette recorder has a tone control then set it to maximum 'treble' and leave it there. Now start the cassette recorder playing by pressing the PLAY button on the recorder. Then adjust the cassette recorder volume control slowly, until you get the message:

```
Your volume control is now properly set. Please
wait while the first program is loaded
```

on the screen. This will give the minimum volume level. You should then increase the setting a little more. If you need to, you can rewind the tape at any time. If no message appears rewind the tape and play it again, increasing the volume control setting in larger steps, or check the cassette leads are correctly plugged in.

The system is very reliable, so if you have problems it may be that your tape recorder is at fault or that you have a fault in the computer. You are advised to contact your dealer.

Note: Each computer program is recorded on the tape as a kind of screeching noise. It's not meant to be listened to, but some cassette recorders have the annoying habit of playing the tape through the loudspeaker while the tape is loading into the computer. Everything depends on what plugs and sockets are being used. It is possible to stop this on most recorders by inserting a small (3.5mm) jack plug into the socket on the recorder marked EAR. You could insert the ear piece supplied with the recorder if that is more convenient. On other recorders you may have to insert a DIN loudspeaker plug, with no wire connections, into the socket marked LS to turn off the noise. Don't try turning the volume control down because then the computer will not be able to 'hear' the tape either. The important thing to do is to try to disable the loudspeaker as described above.

Make a note of the volume setting on your cassette recorder and always use that setting when playing back the WELCOME cassette. You may need to use a different setting with other tapes that you have purchased or recorded yourself.

On the WELCOME cassette the volume control setting is repeated many times at the beginning of the tape. With practice it is possible to save time by running the tape forward by about two minutes (once the volume control is set) and then begin playing the tape from this point, having first entered the command

```
CHAIN "WELCOME"
```
 RETURN

When the first WELCOME program has loaded into the computer it will clear the screen and give you instructions.

The WELCOME pack includes a booklet which describes not only how to get the programs running but also what each of the programs does.

The keyboard

Anyone who has used a standard typewriter will be familiar with the positions of most of the symbols on the keyboard of the BBC Microcomputer. However, there are a number of special keys which need to be mastered (see figure 3) and these are described below.

If you are a keyboard novice you may find the layout confusing. Don't worry – first of all it is not necessary to be a touch typist to work the computer; secondly, there is a program on the WELCOME cassette which will help you to practice finding the various keys, and most people find that with a little practice they become familiar with them fairly quickly.

Some keys have two symbols engraved on them – we'll call those on the top 'upper case' and those below 'lower case' symbols.

CAPS LOCK

When the machine is switched on, the middle light should be on, telling you that the **CAPS LOCK** key is on. This gives capital letters and lower case symbols and is the most useful state for programming because the computer only recognises commands typed in using capital letters. By pressing the **CAPS LOCK** key once you can switch the light off. Now you get lower case letters and lower case symbols. Press it again and it will be on again.

SHIFT

Whether **CAPS LOCK** is on or off, if you press either of the **SHIFT** keys and hold it down while typing in a character you will get a capital letter or upper case symbol.

Holding down **CTRL** and **SHIFT** together stops the computer 'writing' to the screen. This can be useful if it is writing faster than you can read.

SHIFT LOCK

Pressing this key once gives capital letters and upper case symbols until it is pressed again. It has its own on/off light.

Practice in the use of these keys is given in one of the first programs in the introductory pack – the one called KEYBOARD.

SHIFT CAPS LOCK

Depressing and releasing **SHIFT** and **CAPS LOCK** in unison reverses the effect of the **SHIFT** key, causing a lower case letter to be printed when pressing a letter key with **SHIFT** held down. Pressing **CAPS LOCK** again returns the keyboard to normal.

12

Editing keys

Ten user defined
function keys
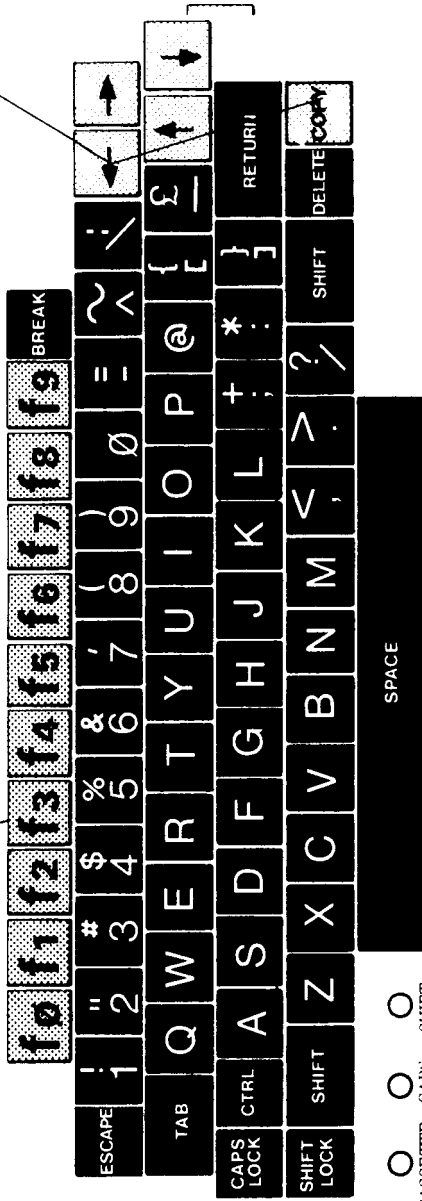
Socket for plug-in
'ROM' cartridges

RETURN

This key is the most commonly used key on the keyboard. When a command or anything else is typed in, it is not usually acted upon until the **RETURN** key is pressed. In other words, this key informs the computer that you have finished entering a line or a reply. Until you press **RETURN**, you can add to or delete what you have typed in.

CURSOR control keys
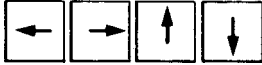
These enable you to move the flashing cursor around the screen when editing a program. Pressing any of them makes the computer automatically enter the 'editing mode' during which two cursors are shown on the screen (see chapter 4).

DELETE

Pressing this key will cause the last character typed in to be erased from the screen. If held down, it will then erase further characters until released.

COPY

This key, used in conjunction with the cursor control keys, enables anything on the screen to be copied – a useful feature when editing a line in a program.

ESCAPE

This key is usually used to stop a program which is running. However, it can be programmed to do other things when pressed – such as moving you from one part of a program to another.

BREAK

This key stops the computer no matter what it is doing. The computer forgets almost everything that it has been set to do. Pressing **BREAK** also resets the screen to `MODE 7`.

Do not get into the habit of using **BREAK**. The **ESCAPE** key provides a much less violent way of escaping from a program! (See chapter 25 for more details on **BREAK**).

CTRL

This key behaves similarly to the **SHIFT** key in that it can be used to change the character generated by other keys. For example, pressing **CTRL** and `G` (called Control G) makes the internal speaker make a short noise. **CTRL B** is used to turn a printer on and **CTRL C** turns it off. **CTRL N** makes the computer stop at the bottom of each page, etc, etc. More information on control codes is given in chapter 34.

TAB

Another key that is useful in special circumstances – like word processing.

These keys can be somewhat confusing because they seem to generate the wrong characters sometimes. The problem is that there are two international standards for displayed characters (Teletext and ASCII) and the BBC Microcomputer can display either. **MODE 7** generates the Teletext display characters and **MODES 0** to **6** show the ASCII characters. But don't worry, the computer recognises the key correctly regardless of what is displayed on the screen. Here is a table showing all these characters:

On the key	Displayed on the screen	
	in MODE 7	in MODES 0 to 6
~	÷	~
^	↑	^
¦	‖	¦
\	½	\
{	¼	{
[	←	[
}	¾	}
]	→	]

Note that in **MODE 7** a zero is shown as a rather pointed O whereas in all other modes, zeros have a slash – 0 – to help to differentiate them from the letter O. The keyboard is also marked in this way.

2 Commands

There are two ways of getting the computer to do something:

1. Give it *commands* which it can act on immediately. This is what happened when you typed in the lines in chapter 1.

2. Give it a series of *numbered instructions*, often called statements, which it can store in its memory and carry out in sequence when told to do so. A stored series of instructions is called a *program*.

Many of the keywords in BASIC can be used both as commands and as statements in a program.

The rest of this chapter is concerned with 'command mode'.

PRINT is used to make the computer print something on the screen. Try these two examples:

```
PRINT "HELLO"
```

Don't forget to press **RETURN** at the end of each line.

```
PRINT 3 + 4
```

In the second example you have given the computer a command to print the sum of 3 and 4. The computer can very easily do addition, subtraction, multiplication and division. The addition, subtraction, multiplication and division signs areall on the right side of the keyboard. If you are interested in doing mathematical or financial work then you will need to know the symbols that the computer uses for various mathematical operations. They are:

+   Addition

–   Subtraction

*   Multiplication

\   Division

^   Exponentiation

.   Decimal point

If you want to get the **+** or ***** then you will have to press the **SHIFT** key as well as the key you want. It's rather like a typewriter: while holding the **SHIFT** key down, press the **+** sign once.

Try typing in the following and check that they work – in other words see that they produce the expected answers.

```
PRINT 4 + 8
```

```
PRINT 18 - 2 * 4
```

```
PRINT 131/4
```

```
PRINT SQR(2)
```

The last one will print the square root of 2 which is 1.41421356.

Then try

```
MODE 5
```

which will make the computer clear the screen and get it ready to draw lines as well as text. In this mode

```
COLOUR 129
```

will select a red background, and

```
CLS
```

will clear the screen to the background colour. In each case you have given the computer a command and it has obeyed it immediately. Working like this is called 'working in command mode'.

While in this mode you might like to learn how to use the bright red *user defined function keys*. Each of these keys can be used to store a word or several words. For example they could be programmed so that each one selects a different colour. Try this:

```
*KEY 2 COLOUR 2 |M
```

The | shown above is produced by a special key. On the keyboard this key is the third key from the right on the row below the red keys. In **MODE 7** this key produces ‖ on the screen.

Once you have typed that in then every time you press the key marked **f2**, the computer will change to **COLOUR 2** which gives yellow lettering. In a similar way you could program some of the other keys like this:

```
*KEY 0 COLOUR 0 |M
*KEY 1 COLOUR 1 |M
*KEY 3 COLOUR 3 |M
```

Note the exact position of spaces when you type in a command.

Of course red letters don't show up very well on a red background! You will have noticed the `|M` at the end of each line above. That is the code used to get a **RETURN** into the user defined function keys.

If the picture on your television screen is either too far up or too far down the screen, you can move the whole display with the command ***TV**.

*TV 255 will move down one line

*TV 254 will move down two lines

*TV 1 will move up one line

*TV 2 will move up two lines

The movements come into effect next time you press **BREAK** or change mode.

*TV also controls the interlace of the television display. See chapter 43.

3 An introduction to variables

In the last chapter we made the computer do a number of calculations but it was never expected to remember any of the results after it had printed them out. Suppose that you have to calculate the wages for everyone in a company. After you have worked out each person's wage, it would be useful to be able to add them all together, so that in the end you would know the total wage bill. Keeping track of things that vary during a long calculation is done by using *variables*.

Try typing this line into the computer:

```
LET Z=5
```

And now try typing in each of the following lines:

```
PRINT Z+6
```

```
PRINT Z * 12
```

As you will have seen, once we have told the computer that 'Z is 5' it understands that every time we use the letter Z in a sum it has to go and have a look to find out what the value of Z is (5 in this case) and use that number in the arithmetic that we set it to do. Now type in

```
LET Z=7
```
and then try these two lines:

```
PRINT Z+12
```

```
PRINT Z/3
```

As you will gather the value of Z has changed from 5 to 7. In computer jargon Z is called a *numeric variable*. That means that Z can be used to store any number, and you can change the value of Z at any time you want to.

The computer is able to store hundreds of different variables and the variables don't have to be called something as simple as Z; you can call a variable by as long a name as you want. For example you could type

```
MYAGE=30
```

Notice that **MYAGE** was written without any spaces between the word

MY and **AGE**. There are only four restrictions about the names that we give to variables:

1. There must be no spaces in the middle of a variable name.

2. All variable names must start with a letter – however you can add in as many numbers as you want to later on.

3. You must not use punctuation marks (such as exclamation marks and question marks) in the variable name but you can use an underline character.

4. Variable names should not begin with BASIC keywords like **PRINT** and **LET**. One that is particularly easy to use by mistake is the keyword **TO**. However it is permissible to start a variable name with a lower case 'to' because upper and lower case names are different. There is a full list of keywords in chapter 48 and they are described in detail in chapter 33.

To get lower case characters on the screen, make sure that the **CAPS LOCK** is off by depressing it to turn off its light. Now you will get small letters and numbers. Hold the **SHIFT** key down if you want to use capital with lower case letters.

Any of the following variable names are acceptable.

```
LET AGE=38
LET this year=1984
LET lengthOFrod=18
LET CAR_mileage=13280
LET value5=16.1
LET weight4=0.00135
LET chicken2egg3=51.6
```

However the following variable names are illegal.

```
LET Football Result=3          (There's a space.)

LET Who?=6                     (There's a question mark.)

LET 4thvalue=16.3              (Starts with a number.)

LET TODAY=23                   (Starts with TO.)

LET PRINT=1234.56              (PRINT is a reserved word.)
```

You will notice that in all the examples above we have put the word **LET** before the variable name. That gives a clear indication of what is actually happening inside the computer, namely that the numeric variable this_year, in one of the examples, is being given a new value '1984'. The word **LET** is optional and the computer will also accept

```
this_year=1984
```

This shortened version is more frequently used.

4 A simple program

In the previous chapter we have been giving the computer commands which it obeys immediately. The problem with this technique is that you have to wait until the computer has completed one command before you can give it the next one. If the computer takes a long time to work out one of the problems you have set it, then you may have to waste an awful lot of time just sitting there waiting for it. For example if you want your computer to work out the number of £1, £5 and £10 notes that you will need to pay the wages at the end of the week the computer will take a fair time to calculate all the wages before it can sort out the notes required.

The same problem comes up when you take a car into a garage to be serviced. You could for example stand by the mechanic and say 'First of all I want the oil changed' and then you could wait for him to change the oil. When that is completed you could then say 'Now I want you to replace the bulb that has blown in one of the front headlights' and then you could wait for that job to be done. And thirdly you might say 'The exhaust is making a noise, so I want you to put the car up on the ramp and check it'.

You would spend a great deal of time waiting for the mechanic to complete each job before assigning the next. There is a far more efficient way of doing things; when you go into the garage you give the mechanic a whole set of instructions, for example:

– First of all change the oil.
– Secondly replace the headlight bulb.
– Thirdly stop the noise in the exhaust.

Once you have given your set of instructions and checked that the garage understands what has to be done, you can walk off and have a cup of coffee and then go back expecting the job to be finished. Now the same thing applies with a computer. It is far better to give it a whole set of instructions and let it run while you wander off and have a cup of coffee. 'Writing a computer program' is nothing more than giving a set of instructions.

If you give the computer a command like

```
PRINT "HOW ARE YOU"
```

then the computer will do that immediately. On the other hand, if you give the computer a statement

```
10 PRINT "HOW ARE YOU"
```

then the computer will regard that as instruction number 10 and it will not do it immediately, but expect other instructions to follow. Instruction number 10 is usually referred to as line 10. Again: if there is a line number then the statement is part of a program; if there is no line number then it is a command which the computer must obey immediately.

When you have given the computer a set of instructions and you then want it to carry them out, you type the word **RUN** on the keyboard. The computer will then carry out the instructions that you asked it to do one at a time and in line-number order. In other words, it will 'execute' the program that you have typed in. Just to check that you have got the idea of what is going on, here is a small program that you can type in.

```
10 REPEAT
20 PRINT "GIVE ME A NUMBER";
30 INPUT B
40 PRINT "12 TIMES ";B;" IS ";12*B
50 UNTIL B=0
```

When you **RUN** the program line 20 will print the message

GIVE ME A NUMBER

on the screen.

Line 30 will print a question mark on the screen and wait for you to type in a number (followed by **RETURN** – as usual). The number you type in will become the value of the variable 'B'.

Line 40 will first print the words **12 TIMES** followed on the same line by the number you typed in, followed on the same line by the word **IS** followed by the result of the calculation. The semi-colons tell the computer to print the next item on the same line as the previous one and right up against it.

Line 50 sends the computer back to line 10 unless B=0, when the program will stop.

Another way of stopping the program is to press the 'panic button' which is marked **ESCAPE** (it's at the top left of the keyboard). If the computer seems to be ignoring you because it's too busy running a program. You can nearly always get its attention by pressing the **ESCAPE** key. When you do that it will stop running your program and print a **>** prompt to show that it has stopped the program and is waiting for your command.

When the computer shows a **>** it is in command mode. You can change your program, give it commands for immediate execution, or tell it to **RUN** the program (in its memory) again. It doesn't forget a program when you press **ESCAPE**.

If the computer is in command mode (in other words if the last thing on the screen is **>**) then you can command it to print the program in its memory by typing

LIST

and pressing **RETURN**.

The computer will then give a listing of the program on the screen for you to cheek. If you discover that you have made an error, for example that you have got something wrong in line 20, then it is easy to correct the error. There are two ways of correcting major errors:

– Retype the whole line.
– Use the screen editor.

Using the screen editor

There is a group of six keys on the right hand side of the keyboard which can be used to edit, or alter, program lines that are displayed on the screen. Four of the keys have arrows on them and are coloured a lighter brown that most of the other keys. These keys enable you to move a flashing cursor around the screen to a line that you wish to edit. As soon as you press one of these keys the computer enters a special 'editing mode' where it displays two cursors. The large white block is called the *write cursor* and it shows you where anything that you enter will appear.

The other small, flashing cursor – the *read cursor* – is the one that can be moved around by the arrow keys.
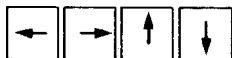
Try moving the read cursor, by using the arrow keys, until it is under a letter at the start of a word and then press the **COPY** key several times. As you will see the **COPY** key copies everything that the read cursor passes under into the new input line. Halfway through copying a line you can always use
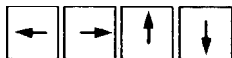
to move the read cursor to some new place on the screen before using **COPY** again to copy some other text to your new input line. The **DELETE** key can always be used to delete characters from the input line.

You can also type new characters in at any time instead of using the **COPY** key. When your new input line is complete just press **RETURN** in the usual way.

Try the following: clear the screen with the command `CLS` and then `LIST` the program. It should include the line

```
20 PRINT "GIVE ME A NUMBER";
```

If not, then type that line in so that you can edit it. Suppose that you wanted to insert the word `BIG` so that line 20 reads

```
20 PRINT "GIVE ME A BIG NUMBER";
```

then all you have to do is to press the up-arrow cursor key until the small flashing line is positioned under the `2` of `20`. Then press the **COPY** key to copy the first part of line 20 to a fresh line at the bottom. When the cursor reaches the space after the `A` where you want to insert the word `BIG`, just type it in with a space in front – it will appear on the bottom line. Then **COPY** the rest of the line 20. The space after the `A` becomes the space after `BIG`. At the end press **RETURN**.

Now try changing the program already in the computer once again by doing the following:

1. List the program by using the `LIST` command.

2. Practice using the cursor control and **COPY** keys to alter line 20 so that it reads:

```
20 PRINT "NOW GIVE ME A BIG NUMBER";
```

3. Now add these new lines. Don't forget to press **RETURN** after each one.

```
5   CLS
25  REPEAT
35  IF B<1000 THEN PRINT "I SAID A BIG NUMBER"
37  UNTIL B>=1000
```

Note: It doesn't matter in what order you type new lines. The computer will automatically put them into numerical order. You will see that this is true by typing

`LIST` **RETURN**

These extra lines tell the computer to reject any number smaller than 1000 and to keep on going back to line 30 to ask for a new number until that number is greater than 1000. The symbol `<` means 'is smaller than', and `>` means 'is greater than'. `IF` and `THEN` are self explanatory.

4. Now `RUN` the program.

```
>RUN
NOW GIVE ME A BIG NUMBER? 16
I SAID A BIG NUMBER
?20
I SAID A BIG NUMBER
?2000
12 TIMES 2000 IS 24000
NOW GIVE ME A BIG NUMBER?
```

This program will go on running until you press **ESCAPE**. If you look you will see that if you give the value 0 for the number, the program never reaches line 50, so it can never end unless you press the panic button!

Deleting part of a program

Quite often you will want to delete a whole line or group of lines in your program. This is easy to do but don't forget that if you type in a new line 20 (for example), it will automatically remove the old line 20 and replace it with your new one. If you want to delete a line completely then type in just the line number and press **RETURN**:

20 RETURN

To delete a whole set of line numbers, for example, lines 50 to 70 inclusive, you can type

DELETE 50, 70

You cannot get these lines back once they are deleted – unless you can copy them off the screen, so use this with care.

After you have deleted several lines – or if you have typed in lots of new lines you often find that you have a very odd set of line numbers. The command

RENUMBER

will make the computer go through your whole program renumbering all the lines so that they are given line numbers in a numeric sequence. Here is an example of terrible programming style – but it will illustrate the **RENUMBER** command. Don't bother to type it in – just look at it.

```
>LIST
  1   REM ** GOTO GOTO GOTO
  2   REM WITH ACKNOWLEDGEMENTS TO
  3   REM "COMPUTERS IN SCHOOLS"
  4   REM THE JOURNAL OF MUSE
 15   GOTO 100
 16   GOTO 95
 40   N=N+1
```

```
 44 END
 57 IF N=18 THEN PRINT "GOTO OR NOT TO GOTO"
 60 IF N>35 THEN GOTO 110
 78 GOTO 40
 95 PRINT "**THE GOTO SHOW**": GOTO 40
100 N=0: GOTO 16
105 PRINT "GOT TO GOTO GOTO NOW"
110 GOTO 44
115 PRINT "GOTO OR NOT TO GOTO";:GOTO 60
>RENUMBER
>LIST
 10 REM ** GOTO GOTO GOTO
 20 REM WITH ACKNOWLEDGEMENTS TO
 30 REM "COMPUTERS IN SCHOOLS"
 40 REM THE JOURNAL OF MUSE
 50 GOTO 130
 60 GOTO 120
 70 N=N+1
 80 END
 90 IF N=18 THEN PRINT "GOTO OR NOT TO GOTO"
100 IF N>35 THEN GOTO 150
110 GOTO 70
120 PRINT "**THE GOTO SHOW**": GOTO 70
130 N=0: GOTO 60
140 PRINT "GOT TO GOTO GOTO NOW"
150 GOTO 80
160 PRINT "GOTO OR NOT TO GOTO";:GOTO 100
>RUN
**THE GOTO SHOW**
```

As you will see, the **RENUMBER** command has not only renumbered the references to line numbers which occur within the program itself – namely after the statements containing the keyword **GOTO**. (This gives the computer the instruction to go to a particular line number and carry out the instruction it finds there.)

Removing a program

If you want to write a new program you will want to remove the old program from the computer's memory. This can be done by using the command **NEW**, or by pressing the **BREAK** key. In either case, if you regret having lost your program, type **OLD** and press **RETURN** and, *providing you haven't begun to type in the new program*, the old one should reappear.

You can always check what's in the memory by typing **LIST**. Try experimenting with these various commands on the program you have typed in.

5 Recording programs on cassette

The WELCOME cassette supplied with your BBC Microcomputer has a number of programs stored on it. You can store a copy of any program on cassette and then load it back into the machine at some time in the future. It really is just like recording music onto a cassette – you can then play the cassette back a few days later and the music will still be there.

If you decide that you don't want to keep the computer program that you have saved on cassette then you can just record a new program over the old one in the same way that you can re-use a cassette when recording music. And in the same way that it is very easy to forget where a particular piece of music is recorded on a cassette, so it's very easy to forget where on the cassette you have stored a particular program. It is very strongly suggested that you *use the tape counter* to keep an index of where programs are on cassette. Also *you must leave gaps between programs*. It is easy to let one program run over the start of the next one if they are all squashed close together. If programs do overlap then you will definitely lose one of them. Be warned!

Most short programs will only move the cassette tape counter on 30 or 40 positions but play safe and spread the programs out over the length of the cassette. If you record the first program at 0000, the second at 0100, the next at 0200 and so on then they will be easy to find and they are unlikely to run over each other.

Note: don't make the mistake of trying to record on the clear plastic tape 'leader' – wind the tape on by hand until the brown tape itself is exposed.

Saving a program on cassette

If you have typed a program into your microcomputer then all you have to do to save it is to

1. Insert the cassette into the recorder.

2. Set the tape counter to 0000 when the tape is fully re-wound.

3. Type

```
SAVE "MYPROG"
```

on the computer and then press the **RETURN** key.

4. The message `RECORD then RETURN` will appear.

5. Fast forward the cassette to the place where you want to record the program - this will be 100 or 200 or 300 etc, on the tape counter.

6. Press the RECORD button on the cassette and then press the **RETURN** key.

If you want to give up at any time then press the **ESCAPE** key.

Notice that `MYPROG` *is the name that we happened to give to the program. You can call your program by any name you like so long as it has no more than ten characters. For example you could have typed*

`SAVE "FRED"` or

`SAVE "GAME3"` or

`SAVE "picture"`

While the program is being saved on cassette the name of the program and some numbers will appear to tell you that things are happening. When the computer has finished, the `>` prompt will re-appear and the tape will stop automatically. If you don't have cassette motor control then you will have to stop the recorder manually after the `>` prompt re-appears. That's it.

Checking a recording

If you want to check that you have successfully recorded your program on the tape then you can use the `*CAT` command (see below). If your recording failed for any reason you can always re-record it. See chapter 35 if you have problems.

Loading a program from cassette

Loading a program back into the computer is just like playing a particular piece of music which has been recorded on the cassette.

1. Type

`LOAD "MYPROG"`

and then press the **RETURN** key. The message

`Searching`

will appear. Of course if your program is called something else then use the right name, for example

`LOAD "GAME3"` **RETURN**

2. Rewind the cassette to just before the start of your program (which will be at 100 or 200 etc.)

3. Check that the volume and tone control settings are correct – see chapter 1 if you are not sure how to find the correct settings.

4. Start playing the cassette by pressing the PLAY button on the recorder.

When the computer finds any program on the cassette it will show the name of the program on the screen. When it finds the program it is looking for it will print

Loading

to let you know that it is now loading the right program.

When the computer has finished loading the program it will print the

>

prompt. It will also automatically stop the tape if you have automatic motor control, if not then you will have to stop the tape manually.

The program is now in the computer. You can type

RUN **RETURN**

to make it work, as usual.

There is one more useful feature to do with loading and saving programs. Instead of typing

LOAD "MYPROG" **RETURN**

you can type

CHAIN "MYPROG" **RETURN**

This not only loads in the program **MYPROG** but also starts it working as soon as it has loaded. It is normally more convenient to use **CHAIN** than **LOAD**.

Cataloguing a tape

If you forget what programs you have on the tape then you can get a catalogue by typing

*CAT

and then playing the tape, but you'll have to wait until the tape has run through the programs.

What the numbers mean

A typical catalogue looks like this

```
WELCOME     00 0084
INTRO       08 088E
INDEX       0A 0ABA
KEYBOARD    25 2545
```

The file-name is followed by two 'hexadecimal' numbers which give the 'block number'. Each program is recorded as a series of 'blocks'. See chapter 10 for an explanation of hexadecimal numbers.

The last number on the line gives the 'length' of the file.

The action of cataloguing a tape also lets the computer verify the information recorded. If there are errors in any of the data on the tape it will print a message and continue.

The **ESCAPE** key allows you to leave cassette operations whenever you like. If you leave from the middle of a `LOAD` operation you will probably get a `Bad Program` error. Type `NEW` to remove this.

More information about cassette formats, loading errors and files is given in Chapter 35.

6 Sample programs

Most of the rest of this book is concerned with introducing the various parts of the BBC BASIC language which the computer understands and other features of the machine. But first, here are a few complete programs which you can try to type in yourself. They must be typed in accurately and can then be run. If a program fails to run properly, then you probably typed a line in incorrectly – for instance, you may have typed `;` when you should have typed `:` or typed `O` instead of `0`.

Most of the sample programs are too big to fit all of the lines on the screen. If you `LIST` a program you have typed in, for example to check that you have made no mistakes, you may find that the lines you want to look at disappear off the top of the screen. To prevent this you can specify the range of lines you want to be listed. For example

```
LIST 100,200
```

will only list those lines numbered between 100 and 200.

Alternatively you can enter 'paged mode' by pressing **CTRL N** (hold down **CTRL** and press **N**). In this mode the listing will stop after every 'page' and will continue only when you press the **SHIFT** key. Paged mode is switched off by pressing **CTRL O** and you should always remember to do this after you have listed the program.

Typing in programs will help you to get a feel for the keyboard and, if you save them on cassette after you are satisfied that they do run properly, will enable you to start building up a library of them.

Learning to use the computer is a little like learning to drive a car – when you first start you find that there are an enormous number of things to think about all at one time. Many of the things you come across from now on will be bewildering at first, but as you get further into the book and as you gain experience in using BASIC, the various parts of the jigsaw puzzle should begin to fall into place. So don't worry if, for instance, some of the comments about the following programs are difficult to understand at first.

Note: In the program listings which follow, extra spaces have been inserted between the line numbers (10,20, etc) and what follows on each line. This is to improve the readability of the programs. However, although it will do no harm, there is no reason to type in any spaces after the line number. For example in the first program, called POLYGON, when entering line 250, all you need to type is

```
250MOVE 0,0
```

POLYGON

This program draws polygons (many sided shapes) in random colours.

Lines 120 to 180 move to a random place on the screen which will be the centre (origin) of the next shape. Lines 210 to 290 calculate the X and Y coordinates of each 'corner' of the polygon and store the values in two 'arrays' for future use. In addition the shape is filled with black triangles (lines 260 and 290) that make it appear that the new polygon is in front of the older ones. Lines 310 to 370 draw all the lines that make up the polygon.

Lines 50 to 70 set the actual colour of logical colours 1,2 and 3 to red, blue and yellow. You can change these if you wish to use other colours.

```
10 REM POLYGON
20 REM JOHN A COLL
30 REM VERSION 1 / 16 NOV 81
40 MODE5
50 VDU 19,1,1,0,0,0
60 VDU 19,2,4,0,0,0
70 VDU 19,3,3,0,0,0
80 DIM X(10)
90 DIM Y(10)
100
110  FOR C=1 TO 2500
120  xorigin=RND(1200)
130  yorigin=RND(750)
140  VDU29,xorigin;yorigin;
150  radius=RND(300)+50
160  sides=RND(8)+2
170  MOVE radius,0
180  MOVE 10,10
190
200  GCOL 0,0
210  FOR SIDE=1 TO sides
220  angle=(SIDE-1)*2*PI/sides
230  X(SIDE)=radius*COS(angle)
240  Y(SIDE)=radius*SIN(angle)
250  MOVE0,0
260  PLOT 85,X(SIDE), Y(SIDE)
270  NEXT SIDE
280  MOVE0,0
290  PLOT 85,radius,0
300
310  GCOL 0,RND(3)
320  FOR SIDE=1 TO sides
330  FOR line=SIDE TO sides
```

```
340   MOVE X(SIDE), Y(SIDE)
350   DRAW X(line), Y(line)
360   NEXT line
370   NEXT SIDE
380   NEXT C
```

You may like to try this alternative for line 200

```
200 GCOL 0, RND(4)
```

MONTHLY

This program plots a set of 'blocks' on the screen which might represent prices over a 12-month period. In this example the height of the bars is randomly selected at line 170. Lines 180 to 270 then draw a 'solid' bar and the edges are marked in black by lines 290 to 330. Lines 340 and 350 print out one letter representing the month of the year at the bottom of each bar.

Notice that lines 60 and 70 set up two of the function keys. Key **f0** sets the computer to **MODE 7** and then lists the program. Key **f9** can be used to run the program.

```
 10 REM MONTHLY
 20 REM JOHN A COLL
 30 REM VERSION 1 / 16 NOV 81
 50
 60 *KEY 0 "MODE7 |M LIST |M"
 70 *KEY 9 "RUN |M"
 80 M$="JFMAMJJASOND"
 90 C=0
100 MODE 2
110 VDU5
120 VDU 29,0;100;
130
140 FOR X=0 TO 1100 STEP 100
150 GCOL 0,C MOD 7+1
160 C=C+1
170 H=RND(400)+200
180 MOVE X,0
190 MOVE X,H
200 PLOT 85,X+100,0
210 PLOT 85,X+100,H
220 MOVE X+70,H+50
230 MOVE X,H
240 PLOT 85,X+170,H+50
250 PLOT 85,X+100,H
260 PLOT 85,X+170,50
```

```
270 PLOT 85,X+100,0
280 GCOL,0
290 MOVEX,H
300 DRAW X+100,H
310 DRAW X+170,H+50
320 MOVE X+100,H
330 DRAW X+100,0
340 MOVE X+10,50
350 PRINT MID$(M$,C,1)
360 NEXT
370
380 GCOL 4,1
390 MOVE 0,450:PRINT "----------------"
400 VDU4
410 PRINTTAB(3,0)"critical level"
```

The height of each bar is set randomly by the variable H. If you want to put in your own values (data), then type the following extra lines. Line 170 must also be deleted by typing **170** followed by **RETURN**.

```
 50 DIM data(12)
 82 FOR J=1 TO 12
 84 PRINT "Input data for month "MID$(M$,J,1);
 86 INPUT data(J)
 88 NEXT J
 89 INPUT "CRITICAL LEVEL", level
155 H=data(C+1)
390 MOVE 0,level:PRINT"---------------"
```

QUADRAT

This program can be used to solve equations of the form

$$Y=Ax^2 + Bx + C$$

The 'roots of the equation' are printed to two decimal places.

The number of decimal places is set by line 90.

The main program between lines 110 and 170 uses three procedures – one for each of the three types of result. The main program is surrounded by

```
REPEAT
.....
.....
.....
UNTIL FALSE
```

which keeps the program going for ever – or until the **ESCAPE** key is pressed.

Line 170 `PRINT'''` prints three blank lines to separate one set of results from the next.

```
 10 REM QUADRAT
 20 REM JOHN A COLL BASED ON A PROGRAM
 30 REM BY MAX BRAMER, OPEN UNIVERSITY
 40 REM VERSION 1.0 /16 NOV 81
 50 REM SOLVES AN EQUATION OF THE FORM
 60 REM A*X^2 + B*X + C
 70 ON ERROR GOTO 350
 80 MODE 7
 90 @%=2020A
100 REPEAT
110 PRINT "What are the three coefficients ";
120 INPUT A,B,C : IF A=0 THEN 110
130 DISCRIM=B^2-4*A*C
140 IF DISCRIM<0 THEN PROCcomplex
150 IF DISCRIM=0 THEN PROCcoincident
160 IF DISCRIM>0 THEN PROCreal
170 PRINT'''
180 UNTIL FALSE
190 END
200
210 DEF PROCcomplex
220 PRINT "Complex roots X=";-B/(2*A);
230 PRINT " +/- "; ABS(SQR(-DISCRIM)/(2*A)) "i"
240 ENDPROC
250
260 DEF PROCcoincident
270 PRINT "Co-incident roots X=";B/(2*A)
280 ENDPROC
290
300 DEF PROCreal
310 X1=(-B+SQR(DISCRIM))/(2*A)
320 X=2(-B-SQR(DISCRIM))/(2*A)
330 PRINT "Real distinct roots X=";X1;" and X=";X2
340 ENDPROC
350 @%=&90A:REPORT:PRINT" at line "ERL
 >RUN
What are the three coefficients ?1,-1,-2
Real distinct roots X=2.00 and X=-1.00

What are the three coefficients ?3,3,3
Complex roots X=-0.50 +/- 0.87i
```

```
What are the three coefficients ?1,2,1
Co-incident roots X=1.00

What are the three coefficients ?
Escape at line 120
>
```

FOURPNT

This program draws a pattern (lines 80 to 140) and then changes foreground and background colours with a half second pause between each change.

```
 10 REM FOURPNT/DRAWS A PATTERN WITH 4 POINTS
 20 REM JOHN A COLL
 30 REM VERSION 1 /16 NOV 81
 50 MODE 4
 60 VDU 29,640;512
 70
 80 FOR A=0 TO 500 STEP 15
 90 MOVE A-500,0
100 DRAW 0,A
110 DRAW 500-A,0
120 DRAW 0,-A
130 DRAW A-500,0
140 NEXT A
150
160 FOR B=0 TO 7 :REM CHANGE THE COLOUR
170 FOR C=1 TO 3
180 T=TIME :REM WAIT A WHILE
190 REPEAT UNTIL TIME-T>50
200 VDU 19,3,C,0,0,0
210 VDU 19,0,B,0,0,0
220 NEXT C
230 NEXT B
```

TARTAN

This program builds up a changing pattern by overdrawing lines on the screen.

The main program between lines 90 and 140 loops for ever and calls various subroutines as necessary. The use of subroutines with implied `GOTO` (eg line 170) results in a structure which is not easy to follow! It would be better to use 'structures' such as procedures (see chapter 17).

```
 10 REM TARTAN
 20 REM BASED ON RESEARCH MACHINES DEMO
 30 REM VERSION 1.0/16 NOV 81
```

```
 40 MODE 2: REM ALSO WORKS IN MODE 5
 50 R=1: D=1: X=0
 60 Y=RND(800)
 70 MOVE X,Y
 80
 90 REPEAT
100 ON D GOSUB 160,260,350,430
110 IF RND(1000)<10 THEN R=D-1
120 GCOL R,(D*1.7)
130 DRAW X,Y
140 UNTIL FALSE
150
160 X=X+800-Y
170 IF X>1000 THEN 220
180 Y=800
190 D=2
200 RETURN
210
220 Y=800/1000-X
230 X=1000: D=4
240 RETURN
250
260 Y=Y-800+X
270 IF Y<0 THEN 310
280 X=1000: D=3
290 RETURN
300
310 X=1000+Y
320 Y=0: D=1
330 RETURN
340
350 X=X-Y
360 IF X<0 THEN 400
370 Y=0: D=4
380 RETURN
390
400 Y=-X: X=0: D=2
410 RETURN
420
430 Y=Y+X
440 IF Y>800 THEN 480
450 X=0: D=1
460 RETURN
470
```

```
480 X=Y-800
490 Y=804: D=3
500 RETURN
```

PERSIAN

This program produces a pattern by drawing hundreds of lines. Random colours are selected by lines 60 and 70. Line 80 moves the origin (middle) of the picture to the middle of the screen.

```
 10 REM PERSIAN
 20 REM ACORN COMPUTERS
 30 REM VERSION 2/16 NOV 81
 40 MODE 1
 50 D%=4
 60 VDU 19,2,RND(3)+1,0,0,0
 70 VDU 19,3,RND(3)+4,0,0,0
 80 VDU 29,640;400;
 90 J1%=0
100 FOR K%=400 TO 280 STEP -40
110 REPEAT J2%=RND(3): UNTIL J2%<>J1%
120 J1%=J2%
130 GCOL 3,J1%
140 FOR I%=-K% TO K% STEP D%
150 MOVE K%,I%
160 DRAW -K%,-I%
170 MOVE I%,-K%
180 DRAW -I%,K%
190 NEXT
200 NEXT
```

SQR ROOT

This program calculates the square root of a number by repeating a simple operation (line 90 and 200) until the calculated result stays steady. The program also indicates how long the calculation takes.

This program illustrates an important mathematical technique but of course you don't have to work out square roots this way – the function SQR is provided in BASIC (see chapter 33).

```
 10 REM ROOT
 20 REM VERSION 1.0 / 16 NOV 81
 30 REM TRADITIONAL ITERATION METHOD
 40 REM TO CALCULATE THE SQUARE ROOT
 50 REM OF A NUMBER TO 3 DECIMAL PLACES
 60 MODE 7
```

```
  70 ON ERROR GOTO 300
  80 @%=&2030A
  90 REPEAT
 100 count=0
 110 REPEAT
 120 INPUT "What is your number ",N
 130 UNTIL N>0
 140 DELTA=N
 150 ROOT=N/2
 160 T=TIME
 170 REPEAT
 180 count=count+1
 190 DELTA=(N/ROOT-ROOT)/2
 200 ROOT=ROOT+DELTA
 210 UNTIL ABS(DELTA) <0.001
 220 T=TIME-T
 230 PRINT
 240 PRINT "Number ",N
 250 PRINT "Root ",ROOT
 260 PRINT "Iterations",count
 270 PRINT "Time",T/100;" seconds"
 280 PRINT''
 290 UNTIL FALSE
 300 @%=&90A:PRINT:REPORT:PRINT
>RUN
What is your number?34

Number         34.000
Root            5.831
Interations     5.000
Time            0.070 seconds

What is your number?125

Number        125.000
Root           11.180
Interations     6.000
Time            0.080 seconds

What is your number?
```

BRIAN

This program prints a 'path in the grass'.

It is a fine example of a 'non-structured' use of BASIC; you might like to try and 'structure' it.

```
 90  REM BRIAN2
100 REM(C) BRIAN R SMITH 1980
110 REM ROYAL COLLEGE OF ART, LONDON
120 REM VERSION 1.0 /16 NOV 81
130 INPUT "NUMBER OF CYCLES e.g. 1 to 5 ",T
140 INPUT "BACKGROUND SYMBOL e.g. +",D$
150 INPUT "MOTIF(<20 chrs.)",A$
160 INPUT "TEXT AFTER DESIGN",B$
170 CLS
180 F=1
190 READ A,G,S,C,D,N
200 H=(D-C)/N
210 X=0
220 J=1
230 X=X+S
240 Y=SIN(X)
250 Y1=1+INT((Y-C)/H+0.5)
260 I=0
270 I=I+1
280 IF I=Y1 THEN 310
290 PRINT D$;
300 GOTO 420
310 Z=Z+F
320 IF Z>0 THEN 350
330 F=-F
340 GOTO 450
350 IF Z<=LEN(A$) THEN 390
360 F=-F
370 Z=Z-1
380 GOTO 310
390 S$=LEFT$(A$,Z)
400 PRINT S$;
410 I=I+Z
420 IF I<40 THEN 270
430 PRINT
440 GOTO 230
450 J=J+1
460 IF J>T THEN 490
470 Z=Z+1
```

```
480 GOTO 310
490 FOR K=1 TO 39
500 PRINT D$,
510 NEXT K
520 PRINT
530 PRINT B$
540 DATA 0,6.4,0.2,-1,1,20
>RUN
NUMBER OF CYCLES e.g. 1 to 5 ?3
BACKGROUND SYMBOL e.g. +?.
MOTIF(<20 chrs.)?Hello David!!
TEXT AFTER DESIGN?That's all folks
```

SINE

This program draws a sine wave on the screen. The computer can draw dotted lines and the feature is used to fill in one part of the sine wave (line 130).

The computer can also print letters anywhere on the screen not just on a 40 by 25 grid. Lines 190 to 220 print a message in the shape of another sine curve.

```
 10 REM SINE
 20 REM JOHN A COLL
 30 REM VERSION 2 / 16 NOV 81
 50 MODE 4
 60 VDU5
 70 GCOL 0,1
 80 VDU19,1,1,0,0,0
 90 MOVE 16,400
100
110 FOR X=0 TO 320
120 IF X<150 THEN MOVE 4*X+16,400
130 PLOT 21,4*X+16,300*SIN(X/48)+400
140 NEXT
160 GCOL 0,1
170 A$="SINE WAVES ARE FAR MORE INTERESTING . . . .
."
180
190 FOR X=1 TO 39
200 MOVE X*1280/40,300*SIN(X/6)+500
210 PRINT MID$(A$,X,1)
220 NEXT
230
240 VDU4
250 END
```

DOUBLE HEIGHT

Here is an example of an assembly language program embedded within a BASIC program between the two brackets [and] which enables you to type in double height letters on the screen.

```
 10 REM DOUBLE HEIGHT IN TELETEXT
 20 WIDTH 36: MODE 7
 30 VDU 28,0,23,39,0
 40 write=!&20E AND &FFFF
 50 DIM PROG 100
 60 FOR PASS = 0 TO 1
 70 P%= PROG
 80 [
 90 OPT PASS*3
100 CMP#&D : BNE noter
110 PHA :JSR write
120 LDA#&8D : JSR write
140 LDA#&08 : JSR write
150 LDA#&8D : JSR write
160 PLA : RTS
170 .noter CMP #&20 : BCS legal
180 JMP write
190 .legal PHA : JSR write
200 LDA #$0B : JSR write
210 LDA #&08 : JSR write
220 PLA : PHA : JSR write
230 LDA #&0A : JSR write
240 PLA : RTS
250 ]
260 NEXT PASS
270 !&20E=!&20E AND &FFFF0000 OR PROG
280 END
```

Line 270 changes the 'write character' routine indirection vector so that all output is sent to the new routine given above. This routine tests for a 'return' code (line 100) and if it finds one it issues Teletext double height control codes on to the next two lines. Otherwise the routine just prints the characters on two lines one above the other so as to produce a double height character. This routine has a quite different effect in non-Teletext modes. Try it. Press **BREAK** after you have finished with this program.

Before we leave this section, here are a few points about entering lines into BASIC.

1. Control characters, for example **CTRL B**, will only be 'reflected' in BASIC and not entered into any program lines, strings etc.

2. Spaces entered in lines will be preserved, including those at the end of the line. This allows blank lines to be entered eg

`10` space **RETURN**

to separate program sections. Some of the programs above have such blank lines. Because of this you should avoid using **COPY** past the true end of a line.

3. Most keywords can be abbreviated using a full stop, eg `L.` for `LIST`, `SA.` for `SAVE`. See chapter 48 for a list of abbreviations.

7 AUTO, DELETE, REM, RENUMBER

BASIC provides a number of facilities to help the user enter programs into the computer and modify programs already there. As you will know by now, it is usual to use line numbers 10, 20, 30, 40 etc for programs. This leaves gaps where the user can insert extra lines later on – for example, he or she might insert lines 11, 12, 13 and 14. When typing in a line of program the user types in the line number first and then the rest of the line. For example:

```
10 PRINT "THIS IS A PROGRAM"
```

The command `AUTO` instructs the computer to 'assign' the line numbers automatically for the user. As an option you can tell the computer to start assigning lines from any number. Thus `AUTO 300` would make the computer produce line number 300, then 310, then 320, etc. There are other options, too, which are explained in chapter 33.

The command `DELETE` allows the user to delete a group of lines from his or her program. When you are writing a long program you often need to be able to delete a large section of it. The keyword `DELETE` is followed by two numbers which give the first and last lines that you wish to remove.

For example

```
DELETE 150,330
```

would delete all the lines with numbers between 150 and 330.

Single lines can be removed by typing in the line number and pressing **RETURN**.

`REM` is a very useful statement. It enables you to put remarks in your program to remind you (not the computer) what is going on. If you are developing a big program – or loading a simple program that you have not used for some time it is very easy to forget how it works or what it does. Normally people place several `REM`s at the start of a program to give general information and then put a `REM` at major points further down the program.

Once you have entered a program you will very often find that the line numbers are no longer in a numeric sequence. As we have seen the command **RENUMBER** makes the computer go through the whole program changing all the line numbers so that they start at line 10 and increase by 10 for each successive line. When you have finished a program it is a good idea to **RENUMBER** it. If you have a program in the computer try

RENUMBER RETURN

and then **LIST** the program to see the effect. After that try

RENUMBER 900,100 RETURN

and you will see, when you list the program, that the computer has renumbered the whole program but the new version has line numbers starting at 900 and this time increasing by steps of 100.

It is possible to put more than one statement on a line. For example, the two statements

CLS (clear the screen)

and

PRINT "HELLO"

can be put on one line, as long as the individual statements are separated by colons, for example:

CLS : PRINT "HELLO"

You can put as many statements on a line as you like as long as the line has less than about 230 characters. The argument for using 'multiple statement lines' is that it saves some memory space and may make the program work a little faster. But the argument against is that you will notice it becomes much more difficult to follow the program when you list it (see chapter 16).

8 Introducing graphics

Modes, colours, graphics and windows

The BBC Microcomputer can display text and windows in eight different screen modes. Only one mode can be used at a time. When the computer is turned on, and also when the **BREAK** key is pressed, it is in **MODE 7**. **MODE 7** will display text (40 columns and 32 rows) and/or graphics. **MODE 7** differs from all the other modes in many ways and a whole chapter (chapter 28) has been devoted to it. In particular it is not easy to draw lines or triangles in **MODE7** and the colour of the text is changed in a different way. Finally some characters are displayed on the screen differently in this mode – for example the character **[** is displayed as ←.

The description that follows assumes that you are in **MODE 5**. To enter **MODE 5** , simply type

MODE 5 RETURN

Note that pressing **BREAK** will return you to **MODE 7** so avoid using **BREAK** . The 'panic button' is marked **ESCAPE**. If you press this the computer will stop what it is doing and return control to you. **MODE 5** is a four colour mode which means that up to four different colours can be shown on the screen at any time. When you enter **MODE 5** two 'colours' are displayed – white letters on a black background. As you will be aware from earlier chapters the colour of the text can be changed by using the **COLOUR** statement, and since this is a four colour mode you can select from:

COLOUR 0   Black

COLOUR 1   Red

COLOUR 2   Yellow

COLOUR 3   White

The same four colours (black, red, yellow and white) may be selected for the background with the commands:

COLOUR 128 (128+0)   Black

COLOUR 129 (128+1)   Red

COLOUR 130 (128+2)   Yellow

COLOUR 131 (128+3)   White

The colour can be used to change the colour of the text foreground and

background – but not the colour of any graphics: for that you need to use another BASIC keyword – **GCOL**, which stands for Graphics COLour.

Graphics

Now for the graphics: when drawing lines and triangles positions on the screen are given with two numbers (the X and Y coordinates).
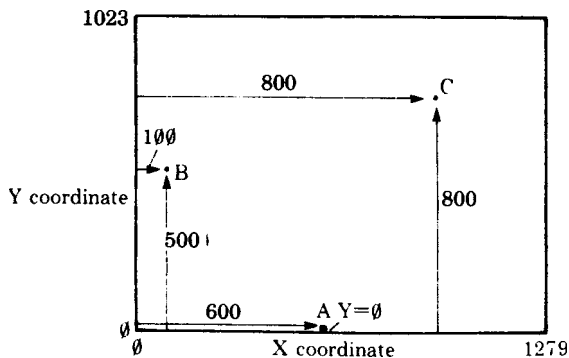
The point A has coordinates 600 across, 0 up. The point B is at position 100,500 and C is at 800,800.

The statement

DRAW 800,800

will draw a line from the last point 'visited' to 600,600. If no point has been visited, the computer will assume that it starts from the point 0,0.

To move without drawing a line use the command **MOVE**. So to draw a line from 1000,0 to 1000,600 type

MOVE 1000,0   RETURN
DRAW 1000,600 RETURN

DRAW 100,500 will draw another line, and so on. As well as **MOVE** and **DRAW** there are **PLOT** commands for other effects. These are described in a later chapter. The statement **GCOL** is used to change the graphics colour used by the **DRAW** statement. **GCOL** is followed by two numbers, the first is normally zero and the second determines the graphics colours, eg:

GCOL 0,0 Black lines

GCOL 0,1 Red lines

GCOL 0,2 Yellow lines

GCOL 0,3 White lines

We'll consider what happens when the first number is not zero later on (chapter 29).

As with the text colours, you can change both foreground and background colours. However, before that can be illustrated it will be easier to set up two windows on the screen – one for text and one for graphics so that you are sure which is which. We will then return to the `GCOL` statement.

Windows

At the moment the whole screen can be used for text and the whole screen can be used for graphics. In some modes (eg `MODE 5`) we can restrict each to a specific window – or section of the screen. In modes without graphics (`MODE 3`, `6` and `7`) only text windows can be used. Imagine we want to create two windows as shown below – on the left a graphics window, on the right a text window. Suppose that the text window stretches from the top of the screen right to the bottom but the graphics window stops short of the bottom:
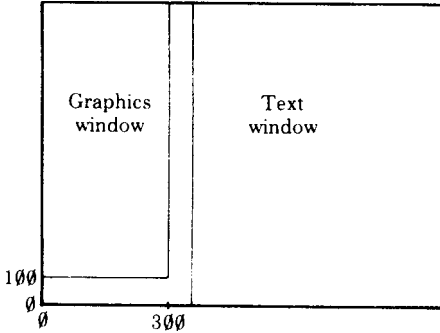
Making a graphics window

Imagine a graphics window which has its edges a, b, c and d 'graphics units' away from the bottom left hand corner of the screen (which is always the starting point for graphics).
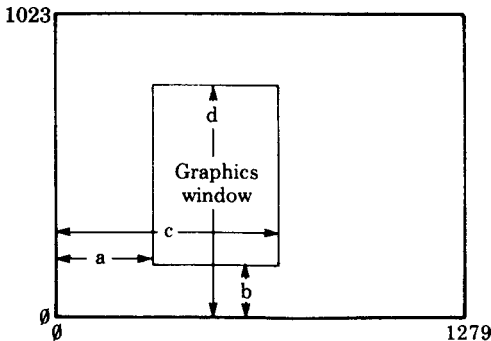
The statement **VDU 24** is used (with some numbers after it) to set up a graphics window (**VDU** stands for Visual Display Unit). For the window shown above the full statement is

```
VDU 24,a;b;c;d;
```

Note: There is a comma after the 24 and a semi-colon after all the other values. The reason for this punctuation is given in chapter 34. So for our actual graphics window we would put

```
VDU 24,0;100;300;799;
```

In all screen modes which can support easily defined graphics the range of values for a, b, c and d is always the same: 0-1023 vertically, 0-1279 horizontally.

Making a text window

Unlike graphics, text 'starts' at the top left hand corner of the screen, so text windows are defined using that point as zero.

Imagine the text window has edges a, b, c and d 'text units' away from the top left of the screen, as shown:
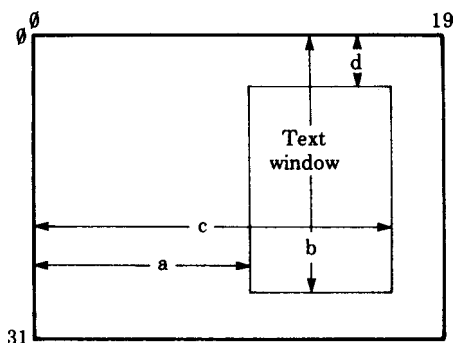
The statement **VDU 28** is used to set up the window as follows:

```
VDU 28,a,b,c,d
```

Note: There is a comma after the **28** and between the other values. There is no comma at the end.

For the text window we wanted to set up, the statement would be

```
VDU 28,5,24,19,0
```

To prove that you now have two separate windows try

```
COLOUR 129
CLS
```

to fill the text window with red and

```
GCOL 0,130
CLG
```

to fill the graphics window with yellow.

Note: In the various different screen modes the number of text characters which can be accommodated along the screen and down the screen is also different. This affects the range of values for the horizontal distances a and c as follows:

MODEs **0** and **3**   (80 characters to a line) 0 to 79
MODEs **1**, **4**, **6** and **7**   (40 characters to a line) 0 to 39

MODEs **2** and **5**   (20 characters to a line) 0 to 19

Similarly the values of b and d depend on the MODE:

MODEs **0**, **1**, **2**, **4** and **5** have 32 lines (0 to 31)

MODEs **3**, **6** and **7** have 25 lines (0 to 24)

To recap, to set up the windows press **BREAK** then type the following – with **RETURN** at the end of each line. You are working in command mode rather than writing a program, so the computer acts on each instruction as you press **RETURN**. It also means that pressing **BREAK** while you are using windows would destroy the text and graphics windows and send the computer back to **MODE 7**.

```
MODE 5
VDU 24,0;100;300;1000;
VDU 28,5,31,19,0
CLS
```

The command **CLS** clears the text from the screen. Now try typing the following lines:

```
DRAW 0,1000
DRAW 100,1000
DRAW 0,0
DRAW 1000,1000
```

You will find that text is now only appearing in the text window and that graphics are only appearing in the graphics window. If you want to clear the text only, type

CLS **RETURN**

If you want to clear the graphics only, type

CLG **RETURN**

(Normally `CLS` clears the whole screen, but where independent text and graphics areas are defined, `CLS` only clears the text.) You will also notice that although some of the commands have told the computer to draw in areas of the screen outside the graphics window, you will not see this on the screen.

Windows may overlap – in fact when you change mode both text and graphics windows fill the whole screen, and you can move windows without destroying what is on the screen, although changing mode does clear the screen. To reset both text and graphics windows to the whole screen, eg in the middle of a program, use `VDU 26`.

`VDU 5` enables text to be drawn at any position inside a graphics window – see chapter 34.

Changing the colours of text and graphics

Now back to text and graphics colours. Let us define the text background to be red and the graphics background to be yellow:

`COLOUR 129` Red text background

`GCOL 0,130` Yellow graphics background

and then clear the text and graphics areas to their background colours:

`CLS` Clear text area

`CLG` Clear graphics area

Now to select the foreground colours for the two areas – for example to obtain yellow letters (text foreground) type `COLOUR 2` and to get black graphics lines type

`GCOL 0,0`

Test this out by typing

`DRAW 150,500`

Although you start up (in `MODE 5`) with the four colours set to black, red, yellow and white, you can select other colours (still of course only four at a time) by using `VDU 19`, as we saw in chapter 1. See chapter 34 for more details of `VDU 19`.

So far we have been working in command mode. Next try typing in this program. You can use `MODE 4` to type the program in but nothing will happen until you run the program. So, press **BREAK** and then the following:

```
 10 MODE 5
 20 VDU 24, 0; 0; 500; 1000;
 30 VDU 28,10,20,19,5
 40 COLOUR 129
```

```
 50 COLOUR 2
 60 GCOL 0,130
 70 CLS: CLG
 80 FOR N = 1 TO 1000
 90 PRINT "LINE"; N
100 GCOL 0, RND(4)
110 DRAW RND(500), RND(1000)
120 NEXT N
>RUN
```

You might like to try saving this program on cassette as described in chapter 5

9 More on variables

In an earlier chapter the idea of 'variables' was introduced. Variables are a fundamental concept in computing, and it is not possible to go far without understanding them.

As we have seen, it is possible to say

```
LET X = 12
```

or just

```
X = 12
```

and the computer knows that it must label a 'box' in its memory with the name X and that the current value of X is 12. With a variable it is possible to alter the value of what is in the box but not the name of the box itself. The statement

```
X = 14
```

simply changes the value of X from 12 to 14. Similarly we can say

```
X = X+1
```

which looks unusual – like an equation which does not balance. In fact all that this is doing is saying to the computer – whatever the value inside your box X, increase it by 1 from now on.

So far we have considered only numeric variables – that is, variables which contain numbers and on which arithmetic can be carried out. But the computer has letters and symbols of various kinds on its keyboard – what about them?

Numbers and characters

Although we can talk of the 'number' 22, we can also consider 22 as a pair of characters – in the same way as A, B, C, ?, $ are characters. In computing it is important to be able to distinguish between numbers and characters. Arithmetic can be carried out on numbers but not on characters. To give you an example to show that this is not such an esoteric idea, consider 22. We can divide 22 by 2 and get 11 if 22 is taken to be a number. But if we talked about a train leaving 'Platform 22' the 22 here would be a pair of characters. You cannot, with a great deal of meaning, divide 'Platform 22' by 2 and get 'Platform 11'.

Next it's important to have a look at the other major kind of variable used in computing – one which can hold characters, not numbers. This is called a *string variable*.

String variables

String variables are used to store 'strings of characters' eg words. They can be recognised easily because they always end with a dollar sign. Here are a few examples of string variables containing various strings of characters. Note that these strings *must* be enclosed by quotation marks.

```
X$ = "HELLO"
DAY$ = "SUNDAY 3RD JANUARY"
NAME$ = "ALEX"
```

In the first example `X$` is called a string variable and `HELLO` is called a string. Once `X$` has been set to contain `HELLO` we can use statements like

```
PRINT X$
```

in just the same way as we said earlier.

```
Z = 5
PRINT Z
```

String variables can be used to hold any number of characters between zero (empty) and 255 (full).

`X$ = ""` will empty `X$`

`X$ = "A"` will set `X$` to contain one character

Of course you cannot use ordinary arithmetic on string variables. For example

```
NAME$ = "SUSAN"
PRINT NAME$ / 10
```

does not make sense. You can't divide Susan's name into ten parts. While you can add, subtract, multiply and divide using numeric variables the only similar operation that can be carried out on string variables is that of 'addition'. Thus

```
10 A$ = "TODAY IS "
20 B$ = "SUNDAY"
30 C$ = A$ + B$
40 PRINT C$
>RUN
TODAY IS SUNDAY
```

The importance of understanding string variables cannot be over-emphasised. Later chapters develop this idea.

How numbers and letters are stored in the computer's memory

Each memory location in the computer can be used to store any number between, and including, 0 and 255, and yet some way has to be found to store letters and also very large numbers. A number of codes are used in the computer in much the same way that different groups of people have used different codes to count. Thus the number 1984 can be written as

	MCMLXXXIV	in Roman numerals
or	1984	in decimal Arabic numerals
or	7C0	in hexadecimal Arabic
or	11111000000	in binary

The need to transmit and store letters has produced another set of codes. The letter 'J' is coded in various ways thus

•－－－	in Morse
10001010	in ASCII binary
4A	in ASCII hexadecimal
74	in ASCII decimal

The ASCII (American Standard Code for Information Interchange) is by far the most common code used by computers to represent characters. A complete code table is given in Appendix C.

When you tell the computer

```
A$ = "HELLO"
```

it stores the ASCII codes for the letters in the word **HELLO** in successive memory locations. The fact that they are stored as ASCII codes is really irrelevant as far as the user is concerned, it just works. However, there are times when the user needs to know about the ASCII codes and two functions are provided to convert between characters and ASCII codes.

The function **ASC** converts a character into its ASCII code. Thus

```
PRINT ASC("J")
```

would print 74.

The reverse function, of converting an ASCII code into a character, is performed by **CHR$**.

Thus **PRINT CHR$(74)** would print the letter **J**. In fact, one quite often needs to use **PRINT CHR$**, so there is a further shortened version of that statement. It is **VDU**; **VDU 74** would also print the letter **J**.

Those doing more complicated programming will need to know the exact way that the computer stores strings and numerics in memory. Full information is given at the end of chapter 39.

Real and integer variables

The numeric variables you have met so far are technically known as *real variables*. They can be used to store any number between 170 000 000 000 000 000 000 000 000 000 000 000 000 (1.7×10^{38}) and 0.000 000 000 000 000 000 000 000 000 000 000 001 47 (1.47×10^{-39}) and can include a decimal point. Of course a similar range of negative numbers can be stored too. The problem with real numbers is that they are only stored to nine figure accuracy, although this is generally accurate enough for most purposes. Note that values can be assigned to variables in exponent format, which is especially useful with very large or very small numbers. For example, instead of typing

`A=1500000000000000000000000000000000`,

`A=1.5E31`

could be typed instead. Another type of numeric variable is an *integer variable*. Integer variable names are distinguished by having a percent sign as the last character of the variable name. They can only store whole numbers between -2,147,483,648 and +2,147,483,647.

On the other hand integer variables are held with complete accuracy – so accounting problems can be dealt with the nearest penny in £2M. Arithmetic calculations with integer variables are significantly faster than with real variables. (See chapter 32 for other suggestions for speeding up programs.)

The two integer operators `MOD` and `DIV` are described in chapter 23.

The variables A% to Z% are special in that they are permanently allocated space in memory. Typing `RUN` or `NEW` does not destroy them. As a result the variables A% to Z% can be set in one program and then used in another program later on without losing their values. Of course the values will be lost if the machine is switched off but otherwise they will remain, even if **BREAK** is pressed.

The variables A% to Z% are called the resident integer variables.

Summary

Three main types of variables are supported in this version of BASIC; they are
integer, real and string.

	Integer	Real	String
Example	346	9.847	"HELLO"
Typical variable	A%	A	A$
Name	SIZE%	SIZE	SIZE$
Maximum size	2,147,483,647	1.7×10^{38}	255 characters
Accuracy	1 digit	9 sig figs	-
Stored in	32 bits	40 bits	ASCII values

All variable names can contain as many characters as required and all characters
are used to identify the variable. Variable names may contain capital letters,
lower case letters and numbers and the underline character. Variable names
must start with a letter and must not start with a BASIC keyword.

10 PRINT formatting and cursor control

This chapter describes the `PRINT` statement which is used to put text on the screen or to a printer. It assumes that you understand that a variable (such as X) can be used to hold a number and that a string variable (such as A$) can be used to hold a line of text.

The following program will help to illustrate some of the ideas. Press **BREAK** and then type in the following program.

```
10 X=8
20 A$="HELLO"
30 PRINT X, X, X
```

When this is `RUN` it produces this:

```
>RUN
          8         8         8
```

This shows that commas separating items in the print list (the print list is the list of things to be printed – `X,X,X` in this case) will force items to be printed in columns or "fields" ten characters wide. Numbers are printed at the right hand side of each column whereas words are printed on the left hand side. You can see the difference if we add some lines to the program.

```
10 X=8
20 A$="HELLO"
30 PRINT X,X/2,X/4
40 PRINTA$,A$,A$
>RUN
          8         4         2
HELLO     HELLO     HELLO
field width
```

Field widths in different screen modes

As we said above, the width of each 'field' is automatically set to ten characters when the computer is switched on.

Since the computer can operate in different screen modes, displaying 20, 40 or 80 characters to the line, clearly the number of fields which can be displayed on

the screen will differ depending on the **MODE**. So try typing in a new line and running the program above.

```
5 MODE 5
```

or

```
5 MODE 0
```
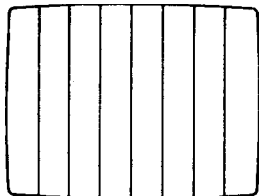

| 80 character modes | 40 character modes | 20 character modes |
| **(MODES 0** and **3)** | **(MODES 1, 4, 6** and **7)** | **(MODES 2** and **5)** |

Note: the widths of the fields can be altered by the use of a special command, **@%** (see below).

Commas between items in the print list always put things in columns or 'fields'. On the other hand semi-colons between items in the print list cause items to be printed next to each other, without spaces:

```
10 X=8
20 A$="HELLO"
30 PRINTA$; X; A$; X; X
>RUN
HELLO8HELLO88
```

Of course if the first item is a number it will be printed to the right of a 'field' unless it is preceded by a semi-colon.

```
10 X=8
20 A$="HELLO"
30 PRINT X; A$; A$

>RUN
          8HELLOHELLO
```

or

```
10 X=8
20 A$="HELLO"
30 PRINT ;X;A$;A$
>RUN
8HELLOHELLO
```

As well as printing variables and string variables as shown above the computer can print any characters placed in between double quotes exactly as they have been typed in, provided they are in a **PRINT** statement. The next program asks for your name and remembers it in the string variable **N$**.

```
10 PRINT "WHAT IS YOUR NAME";
20 INPUT N$
30 PRINT "HELLO";N$;". HOW ARE YOU?"
>RUN
WHAT IS YOUR NAME ?JOHN
HELLO JOHN. HOW ARE YOU?
```

Notice the semi-colon at the end of line 10 that makes the computer stay on the same line while it waits for you to provide it with a value for **N$**. Without the semi-colon this happens:

```
>RUN
WHAT IS YOUR NAME
?JOHN
HELLO JOHN. HOW ARE YOU?
```

Note also the space after the word **HELLO** and before the word **HOW** in line 30. Without these spaces the words run together to produce

```
HELLOJOHN.HOW ARE YOU?
```

It is also legitimate to do calculations in a print list – for example

```
10 X=4.5
20 PRINT X,X+2,X/3,X*X
>
>RUN
        4.5        6.5        1.5        20.25
```

but look what happens if instead of **X=4.5** we put **X = 7**

```
10 X=7
20 PRINT X,X+2,X/3,X*X
>RUN
        7            92.33333333         49
```

because **X/3** is 2.33333333 it makes the number move left in the field until it immediately follows the previous field which contains a 9 and appears to give a result 92.33333333, which is misleading. For this reason, amongst others, the next section is important if you want to print out a lot of numbers.

Altering the width of the field and the way in which numbers are printed

It is often useful to be able to specify the width of the field when printing columns of figures or words and also to be able to specify the number of decimal places to which numbers will be printed.

On the BBC Microcomputer this can be done by setting a special 'variable' (called @%) in a particular way. For the moment this must be treated as a bit of 'magic' but, for example, if we write

```
@%=&20209
```

then this statement tells the computer to print in a field nine characters wide, and that number will be printed with a fixed number of decimal places – in this case, to two decimal places. The following program shows this being used:

```
  5 @%=&20209
10 X=7
20 PRINT X,X+2,X/3,X*X
>RUN
       7.00      9.00      2.33     49.00
```

For the more technically minded

@% is made up of a number of parts.

&	2	02	09
Means hexadecimal numbers follow	Format number 2 ie fixed number of decimal places	Two decimal places characters	Field width of nine

@%=&20309 would give Format 2, three decimal places and field width of nine characters.

```
  5 @%=&20309
10 X=7
20 PRINT X,X+2,X/3,X*X
>RUN
      7.000     9.000     2.333     49.000
```

If you want four decimal places and a field width of 12 you would put the following:

```
5 @%=&2040C
10 X=7
20 PRINT X,X+2,X/3,X*X
```

```
>RUN
    7.0000    9.0000    2.3333    49.0000
```

A few points:

1. The maximum number of significant figures is ten.
2. Format 1 gives figures as exponential values
Format 2 gives figures to a fixed number of decimal places.
Format 0 is the 'normal' configuration.
3. To set the print format back to its initial value (Format 0 and field width ten), set `@%=&90A.`

The `&` tells the computer that the numbers which follow are 'hexadecimal' numbers – that is, numbers based not on 10s but on 16s. Here is a list of hexadecimal numbers (which include the letters A to F).

Decimal number	Hex number
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
10	A
11	B
12	C
13	D
14	E
15	F
16	10
17	11
18	12
19	13
20	14

If you want the computer to print a number or variable in hex then you must put the symbol ~ before it. For example

`PRINT ~12`

will give `C`.

TAB(X)

As well as controlling the print layout by using the comma and semi-colon you can use the **TAB** statement to start printing at a particular place on the screen. You will remember that there can be 20, 40 or 80 characters to the line depending on the **MODE**. **MODE 7** has 40 characters. Try this:

```
10 PRINT "012345678901234567890"
20 F=16
30 REPEAT
40 PRINT TAB(10);F;TAB(15);2*F
50 F=F+1
60 UNTIL F=18
>RUN
012345678901234567890
          16    32
          17    34
```

TAB(10) prints the value of F ten spaces from the left and then **TAB(15)** prints the value of 2*F 15 spaces from the left, on the same line. Note the semi-colon after **TAB(10)** – this causes the computer to begin printing at that point.

Be sure to place an open parenthesis immediately after the word **TAB**. If you leave a space between them, thus: **TAB (10)** the computer will not understand and will report

`No such variable`

If you are beyond the place that you tell the computer to tab to, for example in position 15 with request to **TAB(10)**, then the computer moves to the next line and then tabs ten spaces.
Type in this replacement line:

```
40PRINT TAB(15);F;TAB(10);2*F
>RUN
012345678901234567890
              16
        32
              17
        34
```

TAB(X,Y)

A useful extension of the **TAB** statement allows print to be placed at any specific character location anywhere on the screen. You will remember that in **MODE 7** the text coordinates are
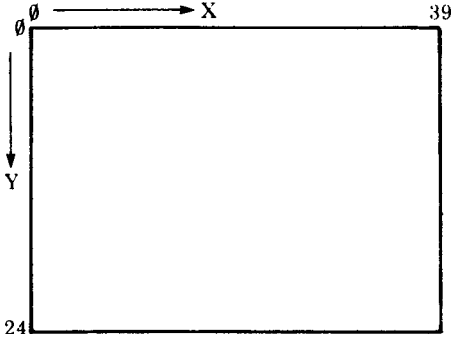
This program counts to 1000, printing as it goes:

```
 5 CLS
10 Q=1
20 REPEAT
30 PRINT TAB(18,5);Q
40 Q=Q+1
50 UNTIL Q=1000
```

The two numbers in parentheses after TAB represent the X and Y text coordinates where printing should start (see also the third program in chapter 23).

Advanced print positioning

Using **PRINT TAB(X,Y)** allows text etc to be printed in any character 'cell' in the appropriate **MODE**. In **MODE 5** there are 20 cells across the screen and 32 cells (lines) down the screen. Sometimes it is useful to be able to position characters on a much finer grid. The statement **VDU5** enables text to be printed at the exact position of the graphics cursor. The statement **MOVE** can be used to position text. *Note that this will not work in MODE 7*. You will remember that the graphics screen is addressed as shown below
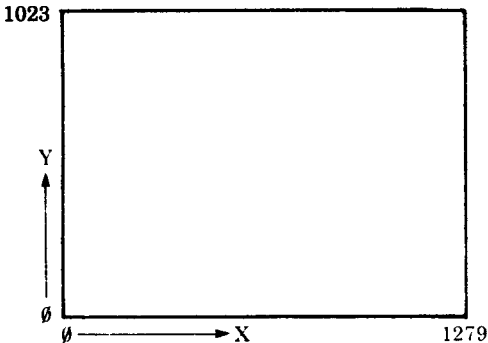
in all modes except **MODE** 7.

Each character cell is 32 graphic units high and, in a 40 character mode such as **MODE 4**, 32 units wide. Suppose we want to subscript a letter to produce for example the chemical formula H_2. This can be done as follows

```
10 MODE 4
20 VDU 5
30 MOVE 500,500
40 PRINT "H";
50 MOVE 532,484
60 PRINT "2"
70 VDU 4
```

Note that the letter **H** is positioned with its top left corner at 500,500. The **2** is then printed one character to the right (532) and a half a character down (484). Again the top left of **2** is at 532,484.

A neater way of achieving the same effect is to replace line 50 with

```
PLOT 0,0,-16
```

One further feature of the BBC Microcomputer which is not normally available on 'personal' computers is the ability to superimpose characters. One obvious use of this facility is to generate special effects such as accents and true underlining. The short program below prints the word **rôle** with the accent correctly placed.

```
10 MODE 4
20 VDU 5
30 X=500
40 Y=500
50 MOVE X,Y
60 PRINT "role"
70 MOVE X+32,Y+16
80 PRINT "^"
90 VDU 4
```

Once in **VDU5** mode the screen will not scroll up when you reach the bottom of the page, instead the writing will start from the top of the screen again. In addition characters will be superimposed on those already on the screen. When in **VDU5** mode you can only print things in the graphics window and not in the text window, and colour is selected with the **GCOL** statement. **VDU5** will not work in text-only modes such as **MODES 3**, **6** and **7**.

Cursor control

The text cursor is the flashing line on the screen which shows where text will appear if it is typed in on the keyboard. The text cursor also indicates where text will be printed on the screen by a **PRINT** statement. The cursor can be moved around the screen by a number of special 'control codes', some of which are as follows.

Code	Effect
8	Move cursor left
9	Move cursor right
10	Move cursor up
11	Move cursor down

These code numbers can be used with either the **VDU** command or the **PRINT** command – eg to move left four spaces, use either

```
VDU 8,8,8,8
```

or

```
PRINT CHR$(8);CHR$(8);CHR$(8);CHR$(8)
```

Clearly the **VDU** command is simpler to type in in most cases.

In addition to the codes shown above the user can use the **PRINT TAB(X,Y)** statement to move the cursor directly to any character position on the screen. As we've seen in **MODE 7** the screen can contain up to 25 lines (numbered 0 to 24) of up to 40 characters per line.
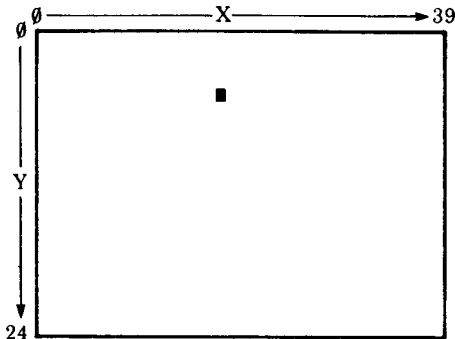
The position marked on the diagram above is 18 positions across and six lines down. The cursor could be moved directly there with the statement

```
PRINT TAB(18,6);
```

Note that the opening parenthesis must immediately follow the word **TAB** thus **TAB(** and not **TAB (**.

Exactly the same effect can be obtained with the statement

```
VDU 31,18,6
```

The cursor can be moved to the 'home' position at the top left of the screen with the statement

```
VDU 30
```

If the user wishes to clear the screen as well as move the cursor to the home position then he or she can use the statement

```
VDU 12
```

The last of the **VDU** commands directly to do with cursor control is **VDU 127** which moves the cursor left and deletes the character there. If you wish to delete the next four characters and then return the cursor to its initial place you could use

```
VDU 9,9,9,9,127,127,127,127
```

Cursor on/off

In some applications the flashing cursor can be a distraction. The cursor can be turned off with the statement

```
VDU 23,1,0;0;0;0;
```

The cursor can be turned back on with the statement

```
VDU 23,1,1;0;0;0;
```

or by changing screen mode using a **MODE** statement.

11 Input

The previous chapter showed how to get information out of the computer and on to the screen. This chapter deals with getting things from the keyboard into the computer. When a program is running there will often be a need for it to request some information from the person at the keyboard.

```
10 PRINT "HOW OLD ARE YOU"
20 INPUT AGE
30 IF AGE<18 THEN PRINT "YOU ARE TOO YOUNG AT ";
40 IF AGE = 18 THEN PRINT "CONGRATULATIONS ON BEING
";
50 IF AGE>18 THEN PRINT "YOU ARE PAST IT IF YOU ARE
";
70 PRINT ;AGE
>RUN
HOW OLD ARE YOU
?22
YOU ARE PAST IT IF YOU ARE 22
```

Line 20 of the above program prints a question mark on the screen and then takes in everything that is typed on the keyboard until **RETURN** is pressed. Line 20 says **INPUT AGE** so the computer is expecting a number since **AGE** is a numeric variable rather than a string variable (see chapter 9). If words are supplied instead of numbers then the computer assumes that the number is zero.

```
>RUN
HOW OLD ARE YOU
?I DON'T KNOW
YOU ARE TOO YOUNG AT 0
```

Because line 20 said **INPUT AGE** a number was expected. If you want to **INPUT** a string (word or group of words) then you must place a string variable (eg **NAME$**) on the input line.

```
10 PRINT "WHAT IS YOUR NAME"
20 INPUT NAME$
30 PRINT "HELLO ";NAME$;" HOW ARE YOU?"
>RUN
WHAT IS YOUR NAME
?JOHN
HELLO JOHN HOW ARE YOU?
```

68

You must follow the word **INPUT** with a numeric variable if you are expecting a number and with a string variable if you are expecting a string.

As you will have seen from the examples above you usually need to print a question on the screen to tell the person at the keyboard what you are waiting for. In the last example the question was 'What is your name'. Instead of placing this in a separate **PRINT** statement you can include the question on the **INPUT** statement.

```
20 INPUT "WHAT IS YOUR NAME ", NAME$
30 PRINT "HELLO ";NAME$;" HOW ARE YOU?"
>RUN
WHAT IS YOUR NAME ? SUSAN
HELLO SUSAN HOW ARE YOU?
```

Notice the punctuation between the question 'What is your name' and the string variable **NAME$**. It is a comma. Notice also that the computer printed a question mark after the question when the program was run. It always prints a question mark on an **INPUT** statement if a comma is used to separate the question from the string variable. If you leave the comma out of the program the computer will leave the question mark out when the program is **RUN**.

```
20 INPUT "WHAT IS YOUR NAME " NAME$
30 PRINT "HELLO ";NAME$;" HOW ARE YOU?"
>RUN
WHAT IS YOUR NAME STEPHEN ALLEN
HELLO STEPHEN ALLEN HOW ARE YOU?
```

The **INPUT** statement, which we have explored above, requires that the user presses the **RETURN** key after he or she has entered the reply. Until the **RETURN** key is pressed the user can delete errors with the **DELETE** key or delete the whole entry so far with **CTRL U**.

Several inputs can be requested at one time. If you type

```
10 INPUT A,B
20 PRINT A,B
```

two numbers will be expected by the computer. They can either be typed in separated by commas, or both can be followed by **RETURN**.

The **INPUT** statement will ignore leading spaces and anything after a comma unless the reply is inside quotation marks.

```
10 INPUT A$
20 PRINT A$
>RUN
?ABC,DEF
ABC
```

The **INPUT LINE** statement can be used in the same way as **INPUT**, but it will accept everything that is typed, including leading spaces and commas. Replace line 10 by

```
10 INPUT LINE A$
```

```
>RUN
?ABC,DEF
ABC,DEF
```

Of course if you make the program

```
10 INPUT A$,B$
20 PRINT A$,B$
```

you will get

```
>RUN
?ABC,DEF
ABC        DEF
```

because now two different inputs are needed in line 10.

12 GET, INKEY

Sometimes it is useful to be able to detect a key as soon as it is pressed without having to wait for the **RETURN** key to be pressed. For example most games react immediately when a key is pressed. There are a group of four functions which respond to single keystrokes.

```
GET
GET$
INKEY
INKEY$
```

The **GET** and **GET$** functions wait until a key is pressed; the **INKEY** and **INKEY$** pair give up after a while if no key is pressed.

```
100 A$ = GET$
```

will wait (for ever) until a key is pressed but

```
100 A$ = INKEY$(200)
```

will wait for only two seconds (200 hundredths of a second). If no key is pressed within two seconds then the computer will move on to the next line of the program and **A$** will be empty. If a key was pressed after say one second then the computer will immediately move on to the next line of the program and will put the character typed into **A$**.

```
100 PRINT "DO YOU WANT TO GO ON"
110 PRINT "YOU HAVE 2 SECONDS TO REPLY"
120 A$=INKEY$(200)
130 IF A$="" THEN PRINT "TOO LATE YOU MISSED IT"
140 IF A$="Y" THEN PRINT "COURAGEOUS FOOL!"
150 IF A$="N" THEN PRINT "COWARD"
```

One of the most common uses of **GET$** is to wait at the bottom of a page for the user to press any key when he or she is ready to go on.

```
100 A$ = GET$
```

GET and **INKEY** are very similar to **GET$** and **INKEY$** but instead of returning a character which can be put into a string variable they return a number which is the ASCII code of the character. The ASCII code of 'Y' is 89 and the ASCII code of 'N' is 78, so the last program could be re-written as

```
100 PRINT "DO YOU WANT TO GO ON"
110 PRINT "YOU HAVE 2 SECONDS TO REPLY"
120 A=INKEY(200)
130 IF A=-1 THEN PRINT "TOO LATE YOU MISSED IT"
140 IF A=89 THEN PRINT "COURAGEOUS FOOL!"
150 IF A=78 THEN PRINT "COWARD"
```

You will see that 'no reply' returns the value -1 when using **INKEY** and returns an empty string when using **INKEY$**.

Advanced features

Another important use of **INKEY** and **GET** is with the group of four direction keys at the top of the keyboard. Normally these are used for editing, but a special statement can make these keys produce ASCII codes like all the other keys on the keyboard. They can then be used by a program for some special purpose – for example to move a point around the screen. The statement ***FX 4,1** makes the editing keys produce ASCII codes and the statement ***FX 4,0** returns the keys to their editing function. The keys produce the following codes:

COPY	135 or (&87)
←	136 or (&88)
→	137 or (&89)
↓	138 or (&8A)
↑	139 or (&8B)

For example:

```
10 *FX 4,1
 20 MODE4
 30 X=500
 40 Y=500
 50 REPEAT
 60 PLOT 69,X,Y
 70 K=GET
 80 IF K=136 THEN X=X-4
 90 IF K=137 THEN X=X+4
100 IF K=138 THEN Y=Y-4
110 IF K=139 THEN Y=Y+4
120 UNTIL Y=0
130 *FX 4,0
```

This program waits at line 70 for a key to be pressed. The program shown above would often be part of a much larger program in which case you would not want everything to stop until a key is pressed. Here it would be better to use **K=INKEY(0)** at line 70 which will let the computer have a quick look to see if a key has been pressed but not wait at all.

```
 10 *FX 4,1
 20 MODE4
 30 X=500
 40 Y=500
 50 REPEAT
 60 PLOT 69,X,Y
 70 K=INKEY(0)
 80 IF K=136 THEN X=X-4
 90 IF K=137 THEN X=X+4
100 IF K=138 THEN Y=Y-4
110 IF K=139 THEN Y=Y+4
120 UNTIL Y=0
130 *FX 4,0
```

13 TIME, RND

TIME

The BBC Microcomputer contains an 'elapsed time' clock. That means that the clock ticks away at a hundred ticks per second but it does not know the real time. However, you can set it and read it. Once set it will stay running until you turn the power off or you do a 'hard reset' (see chapter 25). It can be set to any value, for example 0:

```
TIME = 0
```

This program will print a running stopwatch in the middle of the screen:

```
 5 CLS
10 T = TIME
20 PRINT TAB(10,12);(TIME-T)/100;
30 GOTO 20
```

There is a program to print a 24 hour clock in chapter 23.

RND

When writing games (and simulations), we very often want the computer to make a random choice – or to pick a random number. The most useful function for this is `RND(X)` which picks a random number between 1 and X. The program below prints out a new random number between 1 and 6 every time a key is pressed – like throwing a dice.

```
10 PRINT RND(6)
20 G=GET
30 GOTO 10
```

and this program draws random triangles in random colours

```
10 MODE5
20 PLOT 85,RND(1200),RND(1000)
30 GCOL 0,RND(3)
40 GOTO 20
```

Sometimes it is useful to be able to reset the random number generator to a known value. That may sound a bit strange but when testing a program it is sometimes convenient to have a predictable set of 'random numbers'! To do this the number in parenthesis after the `RND` must be a negative number. Thus `X=RND(-8)` will ensure that the number sequence resulting from `RND` is repeatable.

14 REPEAT...UNTIL, TRUE, FALSE

Computers are fundamentally pretty stupid things but their power comes from their ability to repeat things many times – sometimes many millions of times in one second. In this version of BASIC two types of repeating loops can be used. They are called **REPEAT...UNTIL** and **FOR...NEXT** loops. This chapter explains **REPEAT...UNTIL** loops and the next deals with **FOR...NEXT** loops.

Do you remember the story about a man starting with one grain of rice and doubling it each time he won a bet? How many times would he have to double his grains of rice to own more than a million grains? In the following program **C** is a counter showing how many times the number of grains has doubled and **X** represents the number of grains of rice.

```
10 X=1
20 C=0
30 REPEAT
40 X=X*2
50 C=C+1
60 UNTIL X>1000000
70 PRINT C,X
>RUN
        20    1048576
```

Lines 30 to 60 are called a **REPEAT...UNTIL** loop and everything within the loop is repeated until **X** is greater than one million.

The 'terminating condition' in this program is that **X** is greater than 1000000.

The next program terminates after 15 seconds. Line 40 reads the starting time and the program repeats until the present time minus the starting time is greater than 1500 hundredths of a second – the internal clock ticks a hundred times a second.

```
10 PRINT "SEE HOW MANY SUMS YOU"
20 PRINT "CAN DO IN 15 SECONDS"
30 PRINT
40 STARTTIME=TIME
50 REPEAT
60 F=RND(12)
```

```
 70 G=RND(12)
 80 PRINT "WHAT IS ";F;" TIMES "G;
 90 INPUT H
100 IF H=F*G THEN PRINT "CORRECT" ELSE PRINT "WRONG"
110 PRINT
120 UNTIL TIME-STARTTIME>1500
130 PRINT "TIME UP"
>RUN
SEE HOW MANY SUMS YOU
CAN DO IN 15 SECONDS


WHAT IS 6 TIMES 9?72
WRONG

WHAT IS 1 TIMES 4?4
CORRECT

WHAT IS 9 TIMES 8?72
CORRECT

TIME UP
```

REPEAT...UNTIL loops are very useful and should be used frequently. The next program selects random letters (line 20) and times how long it takes you to find and press the appropriate key. It uses two **REPEAT...UNTIL** loops. One of them is used to wait for a particular key to be pressed on the keyboard.

```
10 REPEAT
20 Z=RND(26)+64
30 PRINT
40 PRINT "PRESS THE KEY MARKED ";CHR$(Z)
50 T=TIME
60 REPEAT UNTIL GET=Z
70 PRINT "THAT TOOK YOU"(TIME-T)/100" SECONDS"
80 UNTIL Z=0
>RUN

PRESS THE KEY MARKED Y
THAT TOOK YOU 1.1 SECONDS

PRESS THE KEY MARKED G
THAT TOOK YOU 1.03 SECONDS
```

Lines 10 and 80 are the main loop and line 60 is a single line **REPEAT**…**UNTIL** loop.

Look at line 80. This will stop the **REPEAT...UNTIL** loop if Z=0. However Z is calculated in line 20 and will have a value between 65 and 90. It will never equal zero, so the program will never stop on its own – you have to press the **ESCAPE** key.

Line 80 says

```
80 UNTIL Z=0
```

Z=0 will never be 'true'. Z=0 will always be 'false', so line 80 can be replaced with

```
80 UNTIL FALSE
```

which just means 'go on for ever'. This is a far better way of doing things than using Z=0 because you might decide to change Z next time you looked at the program. It is also better to use **REPEAT...UNTIL** loops in this way than to put at line 80

```
80 GOTO 20
```

Using **REPEAT...UNTIL** keeps this section of the program well organised. See chapter 19 for a comment on **GOTO**.

If you delete line 10, then the computer will meet an **UNTIL** statement at line 80 with no idea of where the loop is meant to start.

```
>RUN
PRESS THE KEY MARKED A
THAT TOOK YOU 2.09 SECONDS
No REPEAT at line 80
```

In summary **REPEAT...UNTIL** should be used for loops which must terminate on some specific condition.

15 FOR...NEXT

This structure makes the computer repeat a number of statements a fixed number of times. Try the following:

```
10 FOR X = 8 TO 20
20 PRINT X, X+X
30 NEXT X
>RUN
         8          16
         9          18
        10          20
        11          22
        12          24
        13          26
        14          28
        15          30
        16          32
        17          34
        18          36
        19          38
        20          40
```

You can see that the computer looped through line 20 with X taking on the value 8, then 9, then 10 etc up to 20. Each time through the loop, X increased by 1. The 'step size' can be changed easily.

```
10 FOR X = 8 TO 20 STEP 2.5
20 PRINT X, X+X
30 NEXT X
>RUN
         8          16
      10.5          21
        13          26
      15.5          31
        18          36
```

In the two previous examples the value of X (which is called the 'control variable') increased each time through the loop. The 'control variable' can be made to decrease by using a negative step size.

```
10 FOR S = 100 TO 90 STEP -1
20 PRINT S,S/2,S/5
30 NEXT
>RUN
        100            50            20
         99          49.5          19.8
         98            49          19.6
         97          48.5          19.4
         96            48          19.2
         95          47.5            19
         94            47          18.8
         93          46.5          18.6
         92            46          18.4
         91          45.5          18.2
         90            45            18
```

Here is a program which uses several **FOR...NEXT** loops. Some are 'nested' within each other in the way that one **REPEAT...UNTIL** loop was included within another.

```
 10 FOR ROW = 1 TO 5
 20 FOR STAR = 1 TO 10
 30 PRINT"*";
 40 NEXT STAR
 50 FOR STRIPE = 1 TO 20
 60 PRINT "=";
 70 NEXT STRIPE
 80 PRINT
 90 NEXT ROW
100 FOR ROW = 1 TO 6
110 FOR STRIPE = 1 TO 30
120 PRINT"=";
130 NEXT STRIPE
140 PRINT
150 NEXT ROW
```

```
>RUN
**********====================
**********====================
**********====================
**********====================
**********====================
==============================
==============================
==============================
==============================
==============================
==============================
```

The listing shown above is not very easy to follow – try typing

```
LISTO 2
```

and then re-listing the program.

```
>LISTO 2
>LIST
       10 FOR ROW = 1 TO 5
       20   FOR STAR = 1 TO 10
       30     PRINT"*";
       40     NEXT STAR
       50   FOR STRIPE = 1 TO 20
       60     PRINT "=";
       70     NEXT STRIPE
       80   PRINT
       90   NEXT ROW
       100 FOR ROW = 1 TO 6
       110   FOR STRIPE = 1 TO 30
       120     PRINT"=";
       130     NEXT STRIPE
       140   PRINT
       150   NEXT ROW
```

This causes each of the 'nested' **FOR...NEXT** loops to be indented which can make it easier to follow.

Lines 20 to 40 print out ten stars.
Lines 50 to 70 print out 20 equal signs.
Lines 10 and 90 ensure that the above are repeated five times.
Lines 100 to 150 print out six rows of 30 equal signs.

A note on LISTO

LISTO stands for LIST Option and it is followed by a number in the range 0 to 7. Each number has a special effect and details are given in the BASIC keywords chapter under **LISTO**. However, the two most useful values are 0 and 7.

LISTO 0 lists the program exactly as it is stored in memory.

LISTO 1 lists the program with one space after each line number. Most programs in this book have been listed like this.

LISTO 7 lists the program with one space after the line number, and two extra spaces every time a **FOR...NEXT** loop or a **REPEAT...UNTIL** loop is detected.

If you are using the screen editor then make sure that you list the program with **LISTO 0** or else you will copy all those extra spaces into the line!

A few points to watch when using **FOR...NEXT** loops:

1. The loop always executes at least once.

```
10 FOR X=20 TO 0
20 PRINT X
30 NEXT

>RUN
20
```

The loop finishes with the 'control variable' larger than the terminating value. In the next two examples the terminating value is 10.

```
10 FOR Z=0 TO 10 STEP 3
20 PRINT Z
30 NEXT
40 PRINT "OUT OF LOOP"
50 PRINT Z
>
>RUN
         0
         3
         6
         9
OUT OF LOOP
        12

10 FOR Z=0 TO 10 STEP 5
20 PRINT Z
```

```
30 NEXT
40 PRINT "OUT OF LOOP"
50 PRINT Z
>
>RUN
          0
          5
         10
OUT OF LOOP
         15
```

Note that it is not necessary to say **NEXT Z** in line 30: it is optional, though it could be argued that it is clearer to put the Z in.

2. You should *never* jump out of a **FOR...NEXT** loop. It is generally accepted that this is poor style. If you do this your programs will become extremely difficult to follow – there are always better alternatives usually involving the use of a procedure, or setting the control variable to a value greater than the terminating value for example

```
10 FOR X=0 TO 1000
15 PRINT
20 PRINT "TYPE IN A SMALL NUMBER"
30 PRINT "OR ENTER -1 TO STOP THE PROGRAM"
40 INPUT J
50 IF J=-1 THEN X= 2000
60 PRINT "12 TIMES ";J;" IS "; 12*J
70 NEXT X
>
>RUN

TYPE IN A SMALL NUMBER
OR ENTER -1 TO STOP THE PROGRAM
?32
12 TIMES 32 IS 384

TYPE IN A SMALL NUMBER
OR ENTER -1 TO STOP THE PROGRAM
?456
12 TIMES 456 IS 5472

TYPE IN A SMALL NUMBER
OR ENTER -1 TO STOP THE PROGRAM
?-1
12 TIMES -1 IS -12
```

The **REPEAT...UNTIL** loop provides a much better way of dealing with this sort of problem.

3. If you omit the **FOR** statement an error will be generated. First a correct program:

```
10 FOR X=1 TO 5
20 PRINT "HELLO"
30 NEXT
>RUN
HELLO
HELLO
HELLO
HELLO
HELLO
```

and then the program with line 10 deleted

```
20 PRINT "HELLO"
30 NEXT
>RUN
HELLO
No FOR at line 30
```

4. Every **FOR** statement should have a matching **NEXT** statement. This can be easily checked by using **LISTO 7** (list option 7). If the **FOR...NEXT** loops are correctly nested then the **END** in line 50 will line up with the **FOR** in line 5.

```
 5 FOR H=1 TO 4
10 FOR X=1 TO 2
20 PRINT "HELLO" ,H,X
30 NEXT X
40 NEXT H
50 END
>LISTO 7
>LIST
 5 FOR H=1 TO 4
10    FOR X=1 TO 2
20      PRINT "HELLO", H,X
30      NEXT X
40    NEXT H
50 END
>RUN
HELLO                1           1
HELLO                1           2
HELLO                2           1
```

```
HELLO                   2           2
HELLO                   3           1
HELLO                   3           2
HELLO                   4           1
HELLO                   4           2
```

If the **NEXT X** in line 30 is deleted the computer does its best to make sense of the program.

```
 5 FOR H=1 TO 4
10    FOR X=1 TO 2
20    PRINT "HELLO", H,X
40    NEXT H
50 END
>RUN
HELLO                   1           1
HELLO                   2           1
HELLO                   3           1
HELLO                   4           1
```

This is not the way to write programs! Mis-nested **FOR...NEXT** loops will cause problems.

5. In summary **FOR...NEXT** loops should be used when you wish to go through a loop a fixed number of times.

16 IF...THEN...ELSE
More on TRUE and FALSE

The `IF...THEN` statement has been used in several of the programs earlier in this book – for example, in the program in chapter 14 which checked your multiplication. Line 100 was

```
IF H=F*G THEN PRINT "CORRECT" ELSE PRINT "WRONG"
```

As you will realise, this type of statement enables the computer to make a choice as it is working its way through the program. The actual choice that it makes will depend on the values of H, F and G at the time. As a result, the same program can behave in very different ways in different circumstances.

Multiple statement lines

It was explained earlier (chapter 7) that you can put more than one statement on a line and this can be particularly useful with the `IF...THEN` statement. Take, for example:

```
10 X=4 : Y=6 : PRINT "HELLO"
20 PRINT ;X + Y : X=X+Y: PRINT ;X+Y
>RUN
HELLO
10
16
```

which is just the same as

```
10 X=4
20 Y=6
30 PRINT "HELLO"
40 PRINT ;X+Y
50 X=X+Y
60 PRINT ;X+Y
```

This helps to understand how the computer treats multiple statement lines using the `IF...THEN` statement. In the first example which follows, `K=6` and therefore the computer obeys everything after the word `THEN` until the word `ELSE`. Note that a colon only separates statements – the word `ELSE` must be found if you want the other course of action to follow.

```
10 K=6
20 IF K=6 THEN K=9: PRINT "K WAS 6"
ELSE PRINT "K WAS NOT 6": PRINT "END OF LINE"
>RUN
K WAS 6
```

(Note that line 20 was so long that it overflowed on the printer but it is all part of line 20.)

Changing line 10 to `K=7` causes the computer to execute everything after the `ELSE` and as a result it prints

```
K WAS NOT 6
END OF LINE
```

`IF...THEN` is often used with more complicated conditions involving the words `AND`, `OR` and `NOT`. For example:

```
IF X=5 AND Y=6 THEN PRINT "GOOD"
IF X=5 OR Y=6 THEN PRINT "TOO LARGE"
```

The word `NOT` reverses the effect of a condition, thus

```
IF NOT (X=6) THEN PRINT "X NOT 6"
```

These are powerful features which are easy to use.

For the slightly more advanced

It was explained above that you can use multiple statement lines with `IF...THEN` but this leads to messy programs. It is far better to use procedures if you want a whole lot of things to occur. Thus:

```
100 IF H=F*G THEN PROCGOOD ELSE PROCBAD
```

This helps to keep the program readable which is very important, not just from an aesthetic point of view but from the very practical point that a readable program is much easier to get right!

More on TRUE and FALSE

In chapter 14 the concept of `TRUE` and `FALSE` was introduced. A variable can have a numeric value (eg 6 or 15) or it can be `TRUE` or `FALSE`. In fact this is just playing with words (or perhaps we should say numbers) since the computer understands `TRUE` to have the value -1 and `FALSE` to have the value 0.

```
10 IF 6=6 THEN PRINT "YES" ELSE PRINT "NO"
>RUN
YES
```

This prints **YES** because 6=6 is **TRUE**.

```
 5 H=-1
10 IF H THEN PRINT "YES" ELSE PRINT "NO"
>RUN
YES
```

The above program prints **YES** because H is **TRUE** since it has the value -1.

```
 5 H=0
10 IF H THEN PRINT "YES" ELSE PRINT "NO"
>RUN
NO
```

This program sets **H=FALSE** at line 5 so the program prints **NO**. -1 implies **TRUE** and 0 implies **FALSE**. What about other values of H? In fact all non-zero values (except non-integers between -1 and +1) are regarded as **TRUE**, as the following shows:

```
 5 H=-55
10 IF H THEN PRINT "YES" ELSE PRINT "NO"
>RUN
YES
```

Here are some other peculiar examples:

```
10 G= (6=6)
20 PRINT G
>RUN


        -1
```

because (6=6) is **TRUE**.

```
10 IF 5-6 THEN PRINT "TRUE"
>RUN
TRUE
```

This works because (5-6) is -1 which is **TRUE**.

These tricks are more than academic. They can be very useful – not least when you are trying to fathom out what on earth the computer thinks it is doing!

17 Procedures

The BBC Microcomputer has a very complete version of BASIC – often called 'Extended BASIC' and in addition it includes the ability to define and use procedures and functions. It is probably the first version of BASIC in the world to allow full procedure and function handling. These extremely powerful features enable the user to structure his or her programs easily and in addition provide a real introduction to other computer languages like PASCAL.

A procedure is a group of BASIC statements which can be 'called by name' from any part of a program.

```
 10 REM REACT
 20 REM JOHN A COLL
 30 REM BASED ON AN IDEA BY THEO BARRY, OUNDLE
 40 REM VERSION 1/16 NOV 81
 50 @%=&2020A
 60 ON ERROR GOTO 470
 70 MODE7
 80
 90 PROCINTRO
100 REPEAT
110 PROCFIRE
120 PROCSCORE
130 UNTIL FNSTOP
140 END
150
160 DEF PROCINTRO
170 PRINT "This program tests your reactions"
180 PRINT
190 PRINT "Press the space bar to continue"
200 REPEAT UNTIL GET=32
210 CLS
220 ENDPROC
230
240 DEF PROCFIRE
250 CLS
260 PRINT "Press the space bar"
270 PRINT "as soon as a cross appears"
280 T=TIME
290 R=RND(200)+100
300 REPEAT UNTIL TIME>T+R
```

```
310 PRINT TAB(17, 10);"+"
320 *FX 15,1
330 REPEAT UNTIL GET=32
340 DELAY=TIME-T-R
350 ENDPROC
360
370 DEF PROCSCORE
380 PRINT TAB(0, 22);
390 PRINT "You took "; DELAY/100;" seconds"
400 ENDPROC
410
420 DEF FNSTOP
430 PRINT "Do you want another go?"
440 REPLY$=GET$
450 =(REPLY$="N") OR (REPLY$="n")
460
470 @%=&90A
```

The program above shows how named procedures and functions can be used. The main part of the program is between line 90 and line 140.

```
 90 PROCINTRO
100 REPEAT
110 PROCFIRE
120 PROCSCORE
130 UNTIL FNSTOP
140 END
```

The program tests a person's reactions by measuring how long it takes him or her to notice a cross on the screen. As you will see from the section above, line 90 calls a procedure which gives an introduction. The procedure is called **PROCINTRO** and it produces the following on the screen.

```
This program tests your reactions
Press the space bar to continue
```

Then the program repeats **PROCFIRE** and **PROCSCORE** until the user indicates that he or she does not wish to continue.

PROCFIRE produces this:

```
Press the space bar
as soon as a cross appears
+
```

and **PROCSCORE** produces this:

```
You took 2.03 seconds
```

It all seems very straightforward and logical – and it is. Using procedures enables you to split a problem up into a number of small manageable sections and to use (or call) those sections with a sensible name. The main section of most programs should be just a number of procedure calls as are lines 90 to 140. The procedures themselves should be in a separate section – after the **END** statement.

Let us examine **PROCINTRO** more closely.

```
160 DEF PROCINTRO
170 PRINT "This program tests your reactions"
180 PRINT
190 PRINT "Press the space bar to continue"
200 REPEAT UNTIL GET=32
210 CLS
220 ENDPROC
```

Notice how it is defined: line 160 is the start of the definition and the procedure ends at line 220; between those lines are normal BASIC statements. Lines 170, 180 and 190 just print messages on the screen. Line 200 waits until the space bar is pressed, after which line 210 clears the screen.

There are a number of more complex things that can be done with procedures and another program will illustrate the use of parameters – variables passed to the procedure from the main program.

```
 10 REM HYPNO
 20 REM TIM DOBSON / ACORN COMPUTERS
 30 REM VERSION 2 / 16 NOV 81
 40 MODE 5
 50 VDU29,640;512;
 60
 70 FOR X=510 TO 4 STEP -7
 80 GCOL0,X
 90 PROCBOX(X)
100 NEXT
110
120 REPEAT
130 GCOL RND(4),RND(4)
140 FOR K=0 TO 500 STEP 8
150 PROCBOX(K)
160 NEXT K
170 GCOL RND(4),RND(4)
180 FOR K=500 TO 0 STEP -9
190 PROCBOX(K)
200 NEXT K
210 UNTIL FALSE
```

```
220 END
230
240 DEF PROCBOX(J)
250 MOVE -J,-J
260 DRAW -J,J
270 DRAW J,J
280 DRAW J,-J
290 DRAW -J,-J
300 ENDPROC
```

The program uses a procedure called **PROCBOX** which draws a box. The size of the box is determined by the parameter J in the procedure. However you will see that in line 90 the procedure is called with the statement **PROCBOX(X)**. The initial value of X will be 510 because that is the starting value of the **FOR** loop at line 70. This value of X (510)will be passed to the parameter J in the procedure. As a result the procedure **PROCBOX** will draw a box of 'size' 5 10. J is called the 'formal parameter' for the procedure since it is used in the procedure itself. However the X in line 90 and the K in line 150 are referred to as actual parameters. Whatever value K has in line 150 will be transferred to the formal parameter J.

A procedure may have any number of parameters but there must be exactly the same number of actual parameters when the procedure is called as there are formal parameters in the procedure definition. Thus if a procedure was defined like this

```
1000 DEF PROCSWITCH(A,B,C$)


1040 ENDPROC
```

it could not be called with a statement like

```
150 PROCSWITCH(X,Y)
```

but this would be acceptable:

```
150 PROCSWITCH(length, height, NAME$)
```

Local variables in procedures

```
10 J=25
20 FOR X=1 TO 5
30 PROCNUM (X)
40 PRINT "OUT OF PROCEDURE J= ";J
50 NEXT X
60 END
70 DEF PROCNUM(J)
```

```
80 PRINT "IN PROCEDURE J= ";J
90 ENDPROC
>RUN
IN PROCEDURE J= 1
OUT OF PROCEDURE J= 25
IN PROCEDURE J= 2
OUT OF PROCEDURE J= 25
IN PROCEDURE J= 3
OUT OF PROCEDURE J= 25
IN PROCEDURE J= 4
OUT OF PROCEDURE J= 25
IN PROCEDURE J= 5
OUT OF PROCEDURE J= 25
```

In the program above the variable J is used in two ways. The main program starts at line 10 and ends at line 60. The procedure is defined between lines 70 and 90. Line 10 declares that J has the value 25 and the value of J is not changed in the main program. However J is used as the formal parameter in the procedure. All formal parameters are *local* to the procedure which means that their value is not known to the rest of the program. Inside the procedure, J takes on the value of the actual parameter X, but outside the procedure it has a different value. The distinction is made between *global* variables and *local* variables. Global variables are known to the whole program, including procedures, whereas local variables are only known to those procedures in which they are defined and to procedures within that procedure.

In the program above, X is a global varible and it looks as if J is global too, since it is defined in line 10 of the main program. In fact *that* J is global but the use of the parameter J in the procedure creates *another* variable J which is local to the procedure. If a different parameter had been used in the procedure definition then J would have remained global. Thus in the program below the formal parameter has been changed to K in line 70, which leaves J as a global variable.

```
10 J=25
20 FOR X=1 TO 5
30 PROCNUM(X)
40 PRINT "OUT OF PROCEDURE J= ";J
50 NEXT X
60 END
70 DEF PROCNUM(K)
80 PRINT "IN PROCEDURE J= ";J
90 ENDPROC
>RUN
IN PROCEDURE J= 25
```

```
OUT OF PROCEDURE J= 25
IN PROCEDURE J= 25
OUT OF PROCEDURE J= 25
IN PROCEDURE J= 25
OUT OF PROCEDURE J= 25
IN PROCEDURE J= 25
OUT OF PROCEDURE J= 25
IN PROCEDURE J= 25
OUT OF PROCEDURE J= 25
```

The program is pointless in its present form for several reasons – mostly because it doesn't actually do anything with K in the procedure!

Now that J is global its value could be altered anywhere – including inside the procedure. Line 75 increases J by 10:

```
10 J=25
20 FOR X=1 TO 5
30 PROCNUM(X)
40 PRINT "OUT OF PROCEDURE J= ";J
50 NEXT X
60 END
70 DEF PROCNUM(K)
75 J=J+10
80 PRINT "IN PROCEDURE J= ";J
90 ENDPROC
>RUN
IN PROCEDURE J= 35
OUT OF PROCEDURE J= 35
IN PROCEDURE J= 45
OUT OF PROCEDURE J= 45
IN PROCEDURE J= 55
OUT OF PROCEDURE J= 55
IN PROCEDURE J= 65
OUT OF PROCEDURE J= 65
IN PROCEDURE J= 75
OUT OF PROCEDURE J= 75
```

It has been pointed out that all formal parameters are local to the procedure in which they are defined (and to inner procedures) but other variables can be declared as **LOCAL** if required. We very often use the variable X as a counter for a **FOR...NEXT** loop and as a result you have to be careful not to use it twice in the same section of a program. Declaring X as local to a procedure ensures that its use locally will not affect the value of X outside the procedure.

```
10 J=25
20 FOR X=1 TO 5
```

```
30 PROCNUM(X)
40 PRINT "OUT OF PROCEDURE J= ";J
50 NEXT X
60 END
70 DEF PROCNUM(K)
72 LOCAL X
75 FOR X= 1 TO 10
80 J=J + J/X
85 NEXT X
90 ENDPROC
>RUN
OUT OF PROCEDURE J= 275
OUT OF PROCEDURE J= 3025
OUT OF PROCEDURE J= 33275
OUT OF PROCEDURE J= 366025
OUT OF PROCEDURE J= 4026275
```

In the program above, X is used twice – once in the main program (lines 20 and 50) and secondly, and very differently, as a local variable in the procedure. J remains global.

It is wise to declare variables as **LOCAL** in procedures and functions wherever possible except when the variable is a formal parameter. A formal parameter is automatically local and therefore does not need to be declared.

18 Functions

Functions are in many ways similar to procedures but there is one major difference – they always calculate a result which may be a number or a string. BASIC already contains a number of functions. For example the function SQR returns the square root of a number. The square root of 16 is 4 so the statements

```
Y = SQR(16)
```

and

```
PRINT SQR(16)
```

make sense. The first example calculates the square root of 16 and places the result in Y. Compare this to a procedure – for example the one above, to draw a box. The procedure makes things happen (a box appears on the screen) but it does not produce a numeric or a string value. Functions always produce a numeric or string result.

If you have a reasonable understanding of procedures and parameters then you can probably cope with this example of a function:

```
 10 PRINT "GIVE ME THREE NUMBERS ";
 20 INPUT A,B,C
 30 PRINT "THE SUM OF THE NUMBERS IS ";
 40 PRINT FNSUM(A,B,C)
 50 END
100 DEF FNSUM(X,Y,Z)
105 LOCAL K
110 K=X+Y+Z
120 =K
>RUN
GIVE ME THREE NUMBERS ?2,4,4
THE SUM OF THE NUMBERS IS          10
```

Again this program is not of much use – we are using a sledge hammer to crack a nut – but we had better learn to walk before we run!

The function is defined in lines 100 to 120 and three parameters are passed to the function. A, B and C are the actual parameters and the numbers in A, B and C are passed to formal parameters X, Y and Z. For the sake of illustration a local variable K has been used. Line 110 sets K equal to the sum of X, Y and Z. Line 120 shows the way in which a function is ended. It says that the function FNSUM has the value of K.

The example above was spread out to show how a function can be constructed – it could have been compressed to

```
 10 PRINT "GIVE ME THREE NUMBERS ";
 20 INPUT A,B,C
 30 PRINT "THE SUM OF THE NUMBERS IS ";
 40 PRINT FNSUM(A,B,C)
 50 END
100 DEF FNSUM(X,Y,Z)
120 = X+Y+Z
```

or even to the single line function shown below

```
 10 PRINT "GIVE ME THREE NUMBERS ";
 20 INPUT A,B,C
 30 PRINT "THE SUM OF THE NUMBERS IS ";
 40 PRINT FNSUM(A,B,C)
 50 END
100 DEF FNSUM(X,Y,Z) = X+Y+Z
```

Of course we could have managed without a function at all…

```
10 PRINT "GIVE ME THREE NUMBERS ";
20 INPUT A,B,C
30 PRINT "THE SUM OF THE NUMBERS IS ";
40 PRINT A+B+C
50 END
```

…and clearly that would have been the right thing to do in this case. However as soon as your programs reach 40 or 50 lines you should be using procedures extensively and functions occasionally.

As mentioned at the start of this chapter, functions can be used to calculate a numeric or a string result. The function which follows returns the middle letter of a string. The string is passed as a parameter

```
100 DEF FNMID(A$)
110 LOCAL L
120 L=LEN(A$)
140 =MID$(A$,L/2,1)
```

Again, the function is terminated by a statement starting with an equal sign. To use the above function type in the following additional lines.

```
 10 INPUT Z$
 20 PRINT FNMID(Z$)
 30 END
```

Notice that the function is placed after the **END** statement where it will not be executed unless it is called by name.

19 GOSUB

This statement allows the program temporarily to divert to another section. Think about the process of writing a letter. In essence it is really a straightforward procedure – but in practice while the main aim is to write the letter there are often several diversions like the need to get another sheet of paper or answer the phone. These small 'sub-tasks' are essential but if we write a description of every single thing that occurred while writing a letter the reader would probably be so confused that he or she wouldn't realise what the overall aim was. However if the job is described as a series of subroutines or procedures then the main task will emerge more clearly. The subroutine and the GOSUB statement were introduced some years ago to help people who write BASIC programs to break their programs up into recognisable modules. In recent years more flexible and more easily used tools have become available – namely procedures and functions – and these two should be used in preference to GOSUB. None the less, BBC BASIC maintains the GOSUB statement for compatibility with other versions of BASIC.

A temperature scale conversion program is shown in two forms below. Both produce exactly the same output on the computer screen but one has been written using GOSUB and GOTO and the other using procedures.

First with GOSUB and GOTO:

```
 10 REM TEMPERATURE CONVERSION
 20 REM WITHOUT STRUCTURED BASIC
 30 REM THIS IS NOT THE WAY TO WRITE PROGRAMS!
 40 REM JOHN A COLL
 50 REM VERSION 1.0 /22 NOV 81
 60 MODE 7
 70 @%=&2020A
 80 PRINT "ENTER THE TEMPERATURE FOLLOWED BY"
 90 PRINT "THE FIRST LETTER OF THE TEMPERATURE"
100 PRINT "SCALE. e.g. 100C or 72F or 320K"
110 PRINT
120 PRINT "Enter the temperature ";
130 INPUT REPLY$
140 TEMP = VAL(REPLY$)
150 SCALE$=RIGHT$(REPLY$,1)
160 GOODSCALE=FALSE
170 IF SCALE$="C" THEN GOSUB 370
180 IF SCALE$="F" THEN GOSUB 390
```

```
190 IF SCALE$="K" THEN GOSUB 430
200 IF NOT ( GOODSCALE AND TEMP>=-273.16) GOTO 260
210 PRINT''
220 PRINT TEMP; " Celsius"
230 PRINT TEMP+273.16; " Kelvin"
240 PRINT TEMP*9/5 + 32;" Fahrenheit"
250 PRINT
260 IF GOODSCALE THEN 310
270 CLS
280 PRINT "You must follow the temperature with"
290 PRINT "the letter ""C"", ""F"" or ""K"" "
300 PRINT "and nothing else"
310 IF TEMP>=-273.16 THEN 360
320 CLS
330 PRINT "The temperature you have given is"
340 PRINT "too cold for this universe! Try again"
350 PRINT
360 GOTO 110
370 GOODSCALE=TRUE
380 GOTO 460
390 REM CONVERT TO CELSIUS
400 TEMP=(TEMP-32)*5/9
410 GOODSCALE=TRUE
420 GOT0460
430 REM CONVERT TO CELSIUS
440 TEMP=TEMP-273.16
450 GOODSCALE=TRUE
460 RETURN
```

Lines 430 to 460 are referred to as a 'subroutine', and these lines of the program can be called from line 190 by the statement GOSUB 430. Notice that this statement does not give the reader any idea of the purpose of the subroutine. The statement RETURN at the end of the subroutine returns it to the statement after the original GOSUBstatement.

Compare the last program with the one that follows.

```
10 REM TEMPERATURE CONVERSION
20 REM JOHN A COLL
30 REM VERSION 1.0 /22 NOV 81
40 MODE 7
50 @%=&2020A
60 PRINT "ENTER THE TEMPERATURE FOLLOWED BY"
70 PRINT "THE FIRST LETTER OF THE TEMPERATURE"
80 PRINT "SCALE. e.g. 100C or 72F or 320K"
90 REPEAT
```

```
100 PRINT
110 PRINT "Enter the temperature ";
120 INPUT REPLY$
130 TEMP = VAL(REPLY$)
140 SCALE$=RIGHT$(REPLY$,1)
150 GOODSCALE=FALSE
160 IF SCALE$="C" THEN PROCCENT
170 IF SCALE$="F" THEN PROCFAHR
180 IF SCALE$="K" THEN PROCKELVIN
190 PROCEND
200 UNTIL FALSE
210 END
220
230 DEF PROCCENT
240 GOODSCALE=TRUE
250 ENDPROC
260
270 DEF PROCFAHR
280 REM CONVERT TO CELSIUS
290 TEMP=(TEMP-32)*5/9
300 GOODSCALE=TRUE
310 ENDPROC
320
330 DEF PROCKELVIN
340 REM CONVERT TO CELSIUS
350 TEMP=TEMP-273.16
360 GOODSCALE=TRUE
370 ENDPROC
380
390 DEF PROCEND
400 IF GOODSCALE AND TEMP>=-273.16 THEN PROCRESULTS
410 IF NOT GOODSCALE THEN PROCILLEGAL_SCALE
420 IF TEMP< -273.16 THEN PROCILLEGAL_TEMP
430 ENDPROC
440
450 DEF PROCRESULTS
460 PRINT''
470 PRINT TEMP; " Celius"
480 PRINT TEMP+273.16; " Kelvin"
490 PRINT TEMP*9/5 + 32; " Fahrenheit"
500 PRINT
510 ENDPROC
520
530 DEF PROCILLEGAL_SCALE
```

```
540 CLS
550 PRINT "You must follow the temperature with"
560 PRINT "the letter ""C"", ""F"" or ""K"" "
570 PRINT "and nothing else"
580 ENDPROC
590
600 DEF PROCILLEGAL_TEMP
610 CLS
620 PRINT "The temperature you have given is"
630 PRINT "too cold for this universe! Try again"
640 PRINT
650 ENDPROC
```

Obviously the second version is long (about a third longer) but it is much more understandable and this is of crucial importance for medium and large programs.

GOTO

You may have noticed the use of the **GOTO** statement in many of the examples above. **GOTO** is a very useful statement which tells the computer to skip to a particular line number. Beginners in programming find it easy to use. However, it should be used with care because it can lead to what some people call 'spaghetti' programming, a tangle of loops backwards and forwards which makes it very difficult indeed to follow what is going on. The example at the end of chapter 4 shows this in an extreme form.

If you are writing short programs then by all means use **GOTO**. For example, the following program prints out the ASCII code of any key which is pressed – useful if you can't find an ASCII code chart:

```
10 PRINT GET
20 GOTO 10
```

It would be taking things too far to expect people to write

```
10 REPEAT
20 PRINT GET
30 UNTIL FALSE
```

However, when you write programs of more than, say, 50 lines it is a very good idea to try to use the 'structure' provided instead of **GOTO** statements. It is generally accepted that it is still useful to use **GOTO** statements as a last resort when handling error conditions. Use whatever techniques make your program (a) work and (b) easy to follow.

20 ON GOTO, ON GOSUB

There is often a need, in a computer program, to proceed in one of a number of directions. For example your program might present a 'menu' of eight options for the user to choose from. When the user has made the choice your program will need to branch off in the appropriate direction. There are a number of ways of doing this. Here is one in part of a program.

```
100 MODE 7
110 PROCINTRO
120 REPEAT
130 PROCMENU
140 IF M=1 THEN PROCOscar7
150 IF M=2 THEN PROCOscar8
160 IF M=3 THEN PROCUOSAT
170 IF M=4 THEN PROCorbit
180 IF M=5 THEN PROCtransmit
190 IF M=6 THEN PROCshowfigs
200 IF M=7 THEN PROCMercator
210 IF M=8 THEN PROCLocator
220 IF M=9 THEN PROCgetdatetime
230 UNTIL M=-1
240 END
```

Lines 140 to 220 provide exits to a number of procedures all of which will automatically return to the main program. Which procedure is selected depends on the value of M as selected by the user during the procedure **PROCMENU**.

The above method is easy to understand and is recommended but there are other methods which should be noted. The statement **ON...GOTO** also provides a number of exits.

```
100 ON M GOTO 1000,1200,1250,1600
```

would provide an exit to line 1000 of the BASIC program if M=1. If M=2 then control will pass to line 1200 and so on.

An alternative format is

```
100 ON M GOSUB 1000,1200,1350
```

In this case control is passed to the subroutines indicated and then returned to the next line.

Both these techniques are widely used but are less clear than the use of procedures as indicated at the beginning of this chapter.

`ON...GOTO` and `ON...GOSUB` may be used with `ELSE` to trap an `ON` variable which is out of range.

```
60 ON F% GOTO 100,210,350 ELSE PROCfind
```

will perform `PROCfind` if F% is any value other than 1, 2 or 3.

21 Even more on variables

Arrays

Very often we use the computer to store and manipulate sets of data rather than just a single value. For example, we might want to calculate wages for a group of people or sort a group of 20 numbers into order. The 20 numbers might well be associated with 20 names. Arrays make it a lot easier to deal with groups of names and numbers. To get to a more manageable example let's consider working with five names and their associated year of birth. We could store the five names in five variables like this:

```
N1$ = "SARDESON"
N2$ = "MATTINSON"
N3$ = "MOIR"
N4$ = "ALLEN"
N5$ = "MOUNT"
```

That is quite reasonable and it works. If you say

```
PRINT N2$
```

the computer will then print out `MATTINSON`.

However, you cannot tell it to print out the fifth entry or the fourth entry. The computer doesn't have any way of knowing that `N5$` is the fifth entry. Using arrays, though, we can pick out the fifth entry in a long list and that is very useful.

The first thing we have to do is to tell the computer how large an array we are going to use. This is done with a `DIM` statement – eg

```
DIM N$(5)
```

creates an array (a table) and we can then say

```
N$(1) = "SARDESON"
N$(2) = "MATTINSON"
N$(3) = "MOIR"
N$(4) = "ALLEN"
N$(5) = "MOUNT"
```

If we follow that with

```
X = 1
```

and then say

```
PRINT N$(X)
```

the computer will print **"SARDESON"**.

Note that the **X** was a variable which, in this case, had the value of 1.

Here is a complete program – as far as we have got.

```
 10 DIM N$(5)
 20 N$(1)="SARDESON"
 30 N$(2)="MATTINSON"
 40 N$(3)="MOIR"
 50 N$(4)="ALLEN"
 60 N$(5)="MOUNT"
 70 PRINT "WHICH ENTRY DO YOU WANT"
 80 INPUT X
 90 PRINT N$(X)
100 GOTO 70
```

We could also define an array to contain the five years of birth:

```
200 DIM Y(5)
210 Y(1)=1964
220 Y(2)=1960
230 Y(3)=1950
240 Y(4)=1959
250 Y(5)=1962
```

It would be easy to add lines to this program to make the computer search for various things. Of course with only five entries it would undoubtedly be quickest to do the whole thing manually – but with a hundred, a thousand or a million entries the computer would be faster – and certainly more accurate. A few examples of extra lines will make the use of these arrays clearer. Delete lines 70 to 100.

To print out everyone born before 1963

```
300 FOR X=1 TO 5
310 IF Y(X)<1963 THEN PRINT N$(X)
320 NEXT X
```

or to print out everyone whose name contains more than five letters

```
400 FOR X=1 TO 5
410 J$=N$(X)
420 IF LEN (J$)>5 THEN PRINT J$
430 NEXT
```

or to print out everyone whose name begins with M

```
500 FOR X=1 TO 5
510 J$=N$(X)
520 IF LEFT$(J$,1)="M" THEN PRINT J$
530 NEXT
```

All these things can only be done if the computer is able to select a position in a list and it can only do this with arrays.

Note: For an explanation of how the last examples worked, see chapter 22.
You will have noticed that we used the array `N$(X)` to store strings (the names of the people), and array `Y(X)` to store numbers (the years of birth). Each element of the array `N$(X)` can store as long a name as you want (up to 255 characters) and you can dimension `N$` to have as many entries as you want. For example, `DIM N$(1000)` would create a string array with space for 1000 different names. `N$(X)` is called a 'string array' since it is used to store strings.

The array `Y(X)` is called a 'numeric array' and again it can have as many elements (entries) as you need – eg `DIM Y(2000)`. You can also have 'integer numeric arrays' like `DIMJ%(100)`.

As usual on the BBC Microcomputer the story doesn't finish there! There is another whole group of arrays which we haven't met yet. The arrays we have met (both string and numeric) are all 'single-dimension arrays' and could be illustrated by this diagram.

Y(1)	Y(2)	Y(3)	Y(4)	Y(5)
1964	1960	1950	1959	1962

Now suppose we wanted to store the day and month of the birthday as well as the year. We need more boxes.

21	12	4	24	19
2	2	2	10	12
1964	1960	1950	1959	1962

A set of data like that is called a '5 by 3 array' and the (empty) boxes can be set up by the statement

```
10 DIM (5,3)
```

The array could then be filled with the statements

```
20 Y(1,1)=21
30 Y(1,2)=2
40 Y(1,3)=1964
50 Y(2,1)=12
60 Y(2,2)=2
```

```
70 Y(2,3)=1960
```

… etc.

In practice it would involve a lot less typing, and make the program shorter, if all the figures were held in **DATA** statements. You may well need to skip this section at first and return to it when you have understood chapter 22 which deals with the keywords **READ**, **DATA** and **RESTORE**.

If you use **READ** and **DATA** to fill the above five by three array the program could look like this:

```
 10 DIM Y(5,3)
 20 FOR COLUMN=1 TO 5
 30 FOR ROW=1 TO 3
 40 READ Y(COLUMN, ROW)
 50 NEXT ROW
 60 NEXT COLUMN
500 DATA 21,2,1964
510 DATA 12,2,1960
520 DATA 4,2,1950
530 DATA 24,10,1959
540 DATA 19,12,1962
```

The program above takes successive numbers from the **DATA** statements and inserts them into the array. Once this program has been run the array will be set up – filled with the figures – and other sections of the program (not shown above) could search the array as required. The array above is a 'two-dimensional array' used to store numbers. The phrase 'two dimensional' refers to the fact that there are five entries in one dimension and three entries in another dimension – a total of 15 entries. A three-dimensional array could be defined with the statement

```
DIM W(4,5,6)
```

and a four-dimensional array with

```
DIM T(2,2,5,3)
```

This last array would have 2x2x5x3 (60) individual entries. Actually, array elements can be numbered from zero instead of one, so an array declared with

```
DIM V(3)
```

has, in fact, got four elements which are **V(0)**, **V(1)**, **V(2)** and **V(3)**. Similarly the array **T(2,2,5,3)** has 3x3x6x4 (216) elements and will take up over 1000 bytes of memory. Multi-dimension arrays are voracious memory eaters – only use them when needed and, if at all possible, use every element that you set up. There is no limit, other than lack of memory, on the number of dimensions in an array.

At the start of this chapter we set up a string array with the statement **DIM N$(5)**.

This contains six elements, **N$(0)** to **N$(5)**. The length of each string element is limited to the usual 255 characters but you can have as many elements as you wish and as many dimensions – just as for numeric arrays. String arrays are even more ravenous for memory than numeric arrays – use them sparingly!

Just to make sure that the various possibilities are clear, here is a program to set up a string array with first names as well as last names. The program reads names and dates into two arrays:

```
 10 DIM Y(4,2)
 20 DIM N$(4,2)
 30 FOR COLUMN=0 TO 4
 40 FOR ROW=0 TO 2
 50 READ Y(COLUMN, ROW)
 60 NEXT ROW
 70 FOR ROW=0 TO 2
 80 READ N$(COLUMN, ROW)
 90 NEXT ROW
100 NEXT COLUMN
500 DATA 21,2,1964, JAMES,C,SARDESON
510 DATA 12,2,1960,A, MICHAEL, MATTINSON
520 DATA 4,12,1960,CHARLES,C,MOIR
530 DATA 24,10,1959,STEPHEN,R, ALLEN
540 DATA 19,12,1962, GAVIN,,MOUNT
```

22 READ, DATA, RESTORE

One very common way of storing a whole set of information along with the computer program is to use **DATA** statements. You will remember that computer programs can be stored on cassette and sets of data can be stored in the program as well. For example, it might be necessary in a program to convert the month given as a number into a name. The program below stores the names of the month as **DATA**.

```
  5 REPEAT
 10 PRINT "GIVE THE MONTH AS A NUMBER"
 20 INPUT M
 30 UNTIL M>0 AND M<13
 40 FOR X=1 TO M
 50 READ A$
 60 NEXT X
 70 PRINT "THE MONTH IS ";A$
100 DATA JANUARY,FEBRUARY,MARCH,APRIL
110 DATA MAY,JUNE,JULY,AUGUST,SEPTEMBER
120 DATA OCTOBER,NOVEMBER,DECEMBER
>RUN
GIVE THE MONTH AS A NUMBER
?6
THE MONTH IS JUNE
```

Lines 10 to 30 repeat until a sensible value for M is entered – it must be between 1 and 12. In the example run a value of 6 was given to M. In this case the **FOR...NEXT** loop between lines 40 and 60 will repeat six times. Each time through it **READ**s the next piece of **DATA** into **A$** until finally **A$** will be left containing **JUNE**. It might make it clearer if an extra line is temporarily inserted at line 55 to print out the value of **A$** and **X** each time through the loop.

```
>55 PRINT A$,X
>
>LIST
  5 REPEAT
 10 PRINT "GIVE THE MONTH AS A NUMBER"
 20 INPUT M
 30 UNTIL M>0 AND M<13
 40 FOR X=1 TO M
 50 READ A$
```

```
 55 PRINT A$,X
 60 NEXT X
 70 PRINT "THE MONTH IS ";A$
100 DATA JANUARY,FEBRUARY,MARCH,APRIL
110 DATA MAY,JUNE,JULY,AUGUST,SEPTEMBER
120 DATA OCTOBER,NOVEMBER,DECEMBER
>
>RUN
GIVE THE MONTH AS A NUMBER
?6
JANUARY              1
FEBRUARY             2
MARCH                3
APRIL                4
MAY                  5
JUNE                 6
THE MONTH IS JUNE
```

This is one way of getting to the (say) sixth element of a list but there is another way of using an array.

Sometimes there is more than one set of data and it is useful to be able to set the 'data pointer' to a selected set of data. The next program has two sets of data each containing a set of prices and car names. One set of data refers to British Leyland cars and the other to Lotus cars.

```
 10 REPEAT
 20 PRINT "DO YOU PREFER BL OR LOTUS CARS?"
 30 A$=GET$
 40 PRINT A$
 50 IF A$="B" THEN RESTORE 170 ELSE RESTORE 270
 60 INPUT "HOW MUCH ARE YOU WILLING TO SPEND ",P
 80 PRINT "IN THAT CASE, YOU CAN AFFORD THESE:"
 90 FOR X=1 TO 8
100 READ NAME$
110 READ PRICE
120 IF PRICE <P THEN PRINT PRICE,TAB(15); NAME$
130 NEXT X
140 PRINT
150 UNTIL FALSE
160
170 REM BRITISH LEYLAND CARS
180 DATA MINI 1000 CITY, 3198
190 DATA METRO HLE, 4699
200 DATA MAESTRO 1.3L, 5419
```

```
210 DATA MONTEGO 1.6, 5660
220 DATA TRIUMPH ACCLAIM CD, 6239
230 DATA MAESTRO VANDEN PLAS, 7395
240 DATA ROVER 2300S, 10264
250 DATA DAIMLER 4.2, 22995
260
270 REM LOTUS CARS
280 DATA EXCEL, 14990
290 DATA ESPRIT SERIES 3, 15985
300 DATA ESPRIT TURBO, 19980
>RUN
DO YOU PREFER BL OR LOTUS CARS? B
HOW MUCH ARE YOU WILLING TO SPEND ?6000

IN THAT CASE YOU CAN AFFORD THESE:
      3198      MINI 1000 CITY
      4699      METRO HLE
      5419      MAESTRO 1.3L
      5660      MONTEGO 1.6

DO YOU PREFER BL OR LOTUS CARS? L
HOW MUCH ARE YOU WILLING TO SPEND ?1900

IN THAT CASE, YOU CAN AFFORD THESE:

Out of DATA at line 100
```

You will notice that line 50 usees the **RESTORE** statement to set the data 'pointer' to either line 170 where BL data is stored or to line 270 where Lotus data is stored. This ensures that data is read from the correct list.

Lines 90 to 130 attempt to read off eight sets of data from the data lists, but fail when Lotus data is selected as only three sets of data are provided. The message

```
Out of DATA at line 100
```

indicates the failure to find enough entries in the data table. Methods of overcoming the problem are given in chapter 27 which deals with error handling.

23 Integer handling

Two special arithmetical functions are provided which produce integer (ie whole number) results. These integer functions are `DIV` and `MOD` (DIVision and MODulus).

The result of a normal division has two parts – the whole number part and the remainder. Normally the remainder is quoted as a decimal fraction. Thus

11/4 = 2.75 or 2¾

However the functions `DIV` and `MOD` enable the whole number part and the remainder to be calculated separately. Thus

11 DIV 4 = 2

(ie 4 goes into 11 two times) and

11 MOD 4 = 3

(ie the remainder is 3).

A simple division test shows how they can be used.

```
  5 CLS
 10 PRINT "Division test!"
 20 PRINT "Answer with a whole number, and a" '
"remainder"
 30 REPEAT
 40 X=RND(100)
 50 Y=RND(10)
 60 PRINT ' "What is ";X;" divided by ";Y
 70 INPUT A
 80 INPUT "Remainder? "B
 90 IF A=Y DIV Y AND B=X MOD Y THEN PRINT "That's
correct" ELSE PRINT "That's wrong"
100 PRINT ' "Press any key to continue"
110 T=GET
120 UNTIL FALSE
```

`DIV` and `MOD` are used whenever you are trying to convert units – for example seconds into minutes. Thus 500 seconds is **500 DIV 60** minutes and **500 MOD 60** seconds – that is 8 minutes 20 seconds.

For example this program prints a 24 hour clock

```
  5 PRINT "Please input the time"
 10 INPUT "Hours ",H
 20 INPUT "Minutes ",M
 30 TIME=H* 360000 + M* 6000
 40 CLS
 50 REPEAT
 60 SEC=(TIME DIV 100) MOD 60
 70 MIN=(TIME DIV 6000) MOD 60
 80 HR=(TIME DIV 360000) MOD 24
 90 PRINT TAB(7,12) HR;":";MIN;":";SEC;SPC(2)
100 UNTIL FALSE
```

The clock is improved if you type **VDU 23,1,0;0;0;0;** which switches off the flashing cursor (see chapter 10). The next program would keep time to the end of the century – if you left the computer switched on that long!

```
 10 lastminute=0
 20 MODE7
 30 PROCOFF
 40 PROGgetdatetime
 50 CLS
 60 REPEAT
 70 PROCshowtime
 80 UNTIL FALSE
 90 END
100
110 DEF PROCgetdatetime
120 CLS
130 PRINT"Please supply the day, month and year"
140 PRINT "as numbers e.g. 24 10 1984"
150 PRINT
160
170 REPEAT
180 PRINT TAB(5,10);"Day ";
190 INPUT TAB)12,10) ""day
200 UNTIL day>0 AND day<32
210
220 REPEAT
230 PRINT TAB(5,12);"Month ";
240 INPUT TAB(12,12)"" month
250 UNTIL month>0 AND month<13
260
270 REPEAT
280 PRINT TAB(5,14);"Year";
290 INPUT TAB(12,14) "" year
```

```
300 UNTIL year>1799 AND year<2500 OR year>0 AND
year<99
310 IF year<99 THEN year=year+1900
320
330 CLS
340 PRINT"and now the time please"
350 PRINT "using a 24 hour clock"
360
370 REPEAT
380 PRINT TAB(5,10);"Hours ";
390 INPUT hour
400 UNTIL hour>-1 AND hour<24
410
420 REPEAT
430 PRINT TAB(5.12);"Minutes ";
440 INPUT minute
450 UNTIL minute>-1 AND minute<60
460
470 TIME=100*60*(minute+60*hour)
480 ENDPROC
490
500
510 DEF PROCshowtime
520 IF TIME>8640000 THEN
TIME=TIME-8640000
530 hour=TIME DIV 360000 MOD 24
540 minute=TIME DIV (100*60) MOD 60
550 second=TIME DIV 100 MOD 60
560 IF (hour=0 AND minute=0 AND
lastminute=59) THEN PROCincdate
570 lastminute=minute
580 PRINT TAB(0,0);"Date = ";day;" ";
590 RESTORE 600
600 DATA
Jan,Feb,Mar,Apr,May,June,July,Aug,Sept,Oct,Nov,Dec
610 FOR X=1 TO month
620 READ months
630 NEXT X
640 PRINT month$;" ";year;" ";
650 PRINT "GMT = ";
660 IF hour<10 THEN PRINT " ";
670 PRINT;hour;" : ";
680 IF minute<10 THEN PRINT " ";
690 PRINT ;minute; " ";
```

```
700 IF second<10 THEN PRINT " ";
710 PRINT ;second; " "
720 ENDPROC
730
740 DEF PROCOFF
750 VDU 23,1,0;0;0;0;
760 ENDPROC
770
780
790 DEF PROCincdate
800 day=day+1
810 IF (month=2) AND (day>29) THEN day=1:month=3
820 IF (month=2) AND (day=29) THEN IF NOT
FNLEAP(year) THEN day=1:month=3
830 IF ((month=4 OR month=6 OR month=9 OR month=11)
AND (day=31)) THEN day=1: month=month+1
840 IF day>31 THEN day=1:month=month+1
850 IF month>12 THEN month=1:year=year+1
860 ENDPROC
870
880
890 DEF FNLEAP(Y)
900 REM RETURNS TRU IF Y IS LEAP YEAR
910 IF Y MOD 4=0 AND (Y MOD 100<>0 OR Y MOD 400=0)
THEN =TRUE ELSE =FALSE
```

24 String handling

It has been explained that the BBC Microcomputer can store words or other groups of characters in string variables. There are a number of functions which can be used with strings. For example if `A$= "NOTWITHSTANDING"` then the string function `LEFT$` can be used to copy, say, the left three letters of `A$` into another string – `B$`.

```
10 A$="NOTWITHSTANDING"
20 B$=LEFT$(A$,3)
30 PRINT B$
>RUN
NOT
```

Similarly `MID$` can be used to extract the middle section of a string. Change line 20 thus:

```
10 A$="NOTHWITHSTANDING"
20 B$=MID$(A$,4,9)
30 PRINT B$
>RUN
WITHSTAND
```

Line 20 can be read as '`B$` is a copy of the middle of `A$` starting at the fourth letter and continuing for nine letters'. As a result of this flexible use of the word 'middle', `MID$` can in fact be used to copy any part of a string. Change the program again:

```
10 A$="NOTWITHSTANDING"
20 B$=MID$(A$,1,7)
30 PRINT B$
>RUN
NOTWITH
```

As well as `LEFT$` and `MID$` there is the string function `RIGHT$` which copies the rightmost characters of a string.

```
10 A$="NOTWITHSTANDING"
20 B$=RIGHT$(A$,4)
30 PRINT B$
>RUN
DING
```

It is easy to join two strings together to make a long string by using the 'string concatenation operator' which is a plus sign. Its title sounds grand but its purpose is obvious – but quite different from its arithmetic use.

```
10 A$="NOTWITHSTANDING"
20 B$=LEFT$(A$,3)
30 C$=" LIKELY"
40 D$=B$+C$
50 PRINT D$
>RUN
NOT LIKELY
```

The numeric function `LEN` can be used to count up the number of characters in a string – in other words how long it is.

```
10 A$="NOTWITHSTANDING"
20 X=LEN(A$)
30 PRINT X
>RUN
        15
```

`LEN` is very useful if you don't know how long a string is going to be. For example in this palindrome testing program `A$` is copied backwards letter by letter into `B$`.

```
 5 REPEAT
 7 B$=""
10 INPUT "What would you like to reverse? " 'A$
20 FOR T=LEN(A$) TO 1 STEP-1
30 B$=B$ + MID$(A$,T,1)
40 NEXT T
50 PRINT '"If you reverse"' A$'" you get" ' B$
60 UNTIL A$=""
```

The numeric function `INSTR` can be used to see if there is a particular letter (or group of letters) in another string.

```
10 A$="NOTWITHSTANDING"
20 B$="T"
30 X=INSTR(A$,B$)
50 PRINT X
>RUN
         3
```

You will notice that it finds that there is a `T` at position 3 in `A$`. Sometimes it is useful to be able to start the search further along the string. To do this you can add a third parameter which gives that position to start the search.

```
10 A$="NOTWITHSTANDING"
20 B$="T"
30 X=INSTR(A$,B$,4)
50 PRINT X
>RUN
          6
```

If no match is found then **INSTR** returns zero.

```
10 A$="NOTWITHSTANDING"
20 B$="Z"
30 X=INSTR(A$,B4,4)
50 PRINT X
>RUN
          0
```

This looks like the elements of the game called hangman! But first three more string related functions. It is possible to make a string containing many copies of another string by using the string function **STRING$**. So to make a string containing 20 copies of "**ABC**" we write:

```
10 A$="ABC"
20 B$=STRING$(20,A$)
30 PRINT B$
>RUN
ABCABCABCABCABCABCABCABCABCABCABC
ABCABCABCABCABCABCABCABC
```

There is also a function **STR$** which converts a number into a string.

```
10 A=45:B=30
20 A$=STR$(A)
30 B$=STR$(B)
40 PRINT A+B
50 PRINT A$+B$
>RUN
        75
4530
>
```

ie line 40 treats A and B as numbers and line 50 as string characters.

Note that **STR$** is affected by the special variable **@%** if this has been set (see chapter 10).

The opposite function is **VAL**. This extracts the number from the start of the string, which must start with a plus **+** or minus **-** sign or a number. If it

doesn't a zero is returned. Numbers which are embedded in other characters are ignored. So

```
10 A$="124ABC56"
20 PRINT VAL(A$)
```

will print **124**.

However, back to the outline of a hangman program.

```
 10 MODE 7
 20 W=RND(12): REM 12 WORDS TO CHOOSE FROM
 30 FOR X= 1 TO W
 40 READ A$
 50 NEXT X
 60 REM WE HAVE SELECTED A RANDOM WORD
 70 REM NOW GIVE THE USER CHANCES TO
 80 REM GUESS LETTERS IN THE WORD
 90 L=LEN(A$)
100 CORRECT=0
110 TRIES=0
120 PRINT TAB(0,5); "The word has ";L;"letters"
130 PRINT TAB(0,6);"You have ";2*L;"tries"
140 REPEAT
150 PRINT TAB(10,7);"GUESS A LETTER";
160 G$=GET$
170 PRINTTAB(25,7);G$
180 P=0
190 REPEAT
200 P=INSTR(A$,G$,P+1)
210 IF P <>0 THEN PRINT TAB(P+12,15);G$
220 IF P <>0 THEN CORRECT=CORRECT +1
225 IF P=L THEN P=0
230 UNTIL P=0
240 TRIES=TRIES+1
250 PRINT TAB(0,0);"TRIES ";TRIES;
TAB(20,0);"CORRECT ";CORRECT
260 UNTIL (CORRECT=L OR TRIES=2*L)
270 IF CORRECT =L THEN PRINT TAB(10,19);
"Congratulations"
280 IF TRIES=2*L THEN PRINT
TAB(0,16);"The word was ";A$
290 DATA NOTWITHSTANDING
300 DATA INQUISITION
310 DATA MONUMENTAL
320 DATA PRESCRIPTION
```

```
330 DATA CARNIVOROUS
340 DATA TENTERHOOK
350 DATA DECOMPRESSION
360 DATA FORTHCOMING
370 DATA NEVERTHELESS
380 DATA POLICEWOMAN.
390 DATA SOPHISTICATED
400 DATA GUESSTIMATE
```

There are a number of improvements to be made to this program. Its screen layout is poor and also it lets you guess the same letter twice.

It is possible to use some mathematical operators on strings. For example one can check to see if two strings are 'equal' or if one string is 'greater' than another. Obviously the words 'equal' and 'greater' have slightly different meanings when applied to strings. A few examples may help to clarify things. As far as the computer is concerned, 'XYZ' is greater than 'ABC' because X is further down the alphabet than A. Similarly, 'ABC' is greater than 'AB' because 'ABC' is a longer string.

You can use the following comparisons with strings:

=   equal to

<> not equal to

<   less than

>   greater than

<= less than or equal to

>= greater than or equal to

The following are legal statements:

```
IF A$ = "HELLO" THEN PRINT "HOW ARE YOU"
IF B$ > "FIFTEEN" THEN GOTO 1000
```

Notice that if **B$** contained **SIX** it would be regarded as 'greater than' **FIFTEEN** because it starts with an **S** whereas **FIFTEEN** starts with an **F**.

```
10 B$ = "SIX"
20 IF B$ >"EIGHT" THEN STOP
```

This program would stop because the word **SIX** begins with an **S** which is regarded as 'greater than' the letter **E**.

Strings are compared character by character using ASCII codes. If two strings start with an identical sequence of letters, for example **PIN** and **PINT** then the longer string is regarded as the larger one.

25 Programming the red user defined keys

At the top of the keyboard is a group of special red keys which are called *user defined keys*. Instead of producing a fixed character the user can 'define' these keys to generate any character or string of characters that is required. For example, to set up key **f1** so that it produces the word `PRINT` every time it is pressed you can type

`*KEY 1 PRINT` **RETURN**

To set key **f2** to produce the word `DATA` you enter

`*KEY 2 DATA` **RETURN**

If you want to enter more than one word into a user defined key then you can enclose the words in quotes

`*KEY 3 "IF X="` **RETURN**

though quote marks are not necessary.

When you are developing programs it is very useful to have one of the keys set up to change to `MODE 7` and then `LIST` the program automatically. If you were typing in the commands `MODE 7` and `LIST` you would normally follow each with a **RETURN**, and you have to include something equivalent to pressing the **RETURN** key when you set the key up. In fact to set up key **f0** you enter this:

`*KEY 0 MODE 7 |M LIST |M`

The two characters `|` and `M` together are understood to mean the same thing as pressing the **RETURN** key. In fact the `|` in front of any letter makes the computer generate a control character. You may remember that to enter 'paging mode', where the computer stops at the bottom of every page, you can type **CTRL N**. That instruction can be added to the key **f0** definition as well, if you wish.

`*KEY 0 MODE 7 |M |N LIST |M`

It is important to remember than any `*KEY` definition must be the last statement on a line because once the computer finds a `*` at the start of a statement it passes the rest of the line to the Machine Operating System and not to BASIC. The Machine Operating System does not understand `:` which BASIC would understand as a multiple statement separator. The same thing

applies to ***FX** statements – only one is allowed per line.

However, it is acceptable to use colons to separate statements within the key definition. For example:

```
*KEY 6 MOVE 0,0 : DRAW X,Y |M
```

If you want to you can set up the user defined keys in a program in exactly the same way that they are set up in command mode. Thus

```
10 *KEY 7 "|B LIST|M |C"
```

would let key 7 turn the printer on, list the program and then turn the printer off.

If you wish to include an ASCII code greater than 128 (&80) then you can do this by using the sequence **|!** to add 128 to the value produced. For example:

```
*KEY 8 "|!|V"
```

would put a single 'character' in key 8 and the ASCII value of the character would be made up from the two parts. The **|!** is worth 128 and the ASCII value of **CTRL V** is 22, giving a total value of 150.

The BREAK key

Pressing the **BREAK** key causes a 'soft reset' which does not reset the clock or clear the definitions of the user defined keys. However, pressing **BREAK** while the **CTRL** key is pressed will cause a 'hard reset' which resets everything. Pressing **SHIFT** and **BREAK** together is used on disc and runs a program without any further instructions.

As you know, when you press the **BREAK** key the computer is reset and nothing can change that. Your program will stop and all variables will be lost; even your program will appear to be lost. However, there are a number of things that can be done to alter the course of events.

First, a program can be recovered by typing **OLD RETURN** and then **RUN RETURN**. Alternatively, the **BREAK** key can be 'redefined' by using the expression

```
*KEY 10 "OLD |M RUN |M "
```

which treats the **BREAK** key as another user definable key.

Other keys

The ← → ↑ ↓ and **COPY** keys can also be redefined – they can be considered to be user defined keys 11 to 15 (see also chapter 43).

26 Operator priority

An operator is something like **+**, **/**, **<**, etc, which affects one or more items – for example comparing them, or adding them.

Mathematical operators are familiar. Most act on two numbers – for example

3+7	addition
2-5	subtraction
4*6	multiplication
1/9	division
7 DIV 4	integer division
7 MOD 4	integer remainder
3^4	raise to a power

These operators are referred to as binary operators since they require two operands (ie two things to operate on).

-5

This shows one of the few 'unary' operators that we are used to. The **-** just acts on the **5** to make it a negative number.

This version of BASIC has a large number of operators and it is very important that the user is aware of their order of priority. You will remember that in mathematics multiplication must be completed before addition. The same applies to other operators – there is a strict hierarchy and you must be aware of it if the computer is to do what you expect.

The overall order of precedence for operators is as follows.

Group 1	unary minus	
	unary plus	
	NOT	
	functions	
	parentheses **( )**	
	indirection	
	operators (see	
	chapter 39)	
Group 2	**^**	raise to the power
Group 3	*****	multiplication
	/	division
	DIV	integer division
	MOD	integer remainder

Group 4	**+**	addition
	-	subtraction
Group 5	**=**	equal to
	<>	not equal to
	<	less than
	>	greater than
	<=	less than or equal to
	>=	greater than or equal to
Group 6	**AND**	logical and bitwise AND
Group 7	**OR**	logical and bitwise OR
	EOR	logical and bitwise EOR

All operators in each group have equal priority and will be dealt with on a left to right basis – in other words in order in each line.

Some of the operators should be familiar by now, others may need explanation.

Group 1

NOT is most often used to reverse the result of a test, eg

```
IF NOT (X=5) THEN....
```

Clearly this example could be written

```
IF X<>5 THEN
```

but the operator **NOT** is often needed when using functions, eg

```
IF NOT FNVALID THEN...
```

Functions include all the predefined functions such as **SQR,SIN,ASC** etc and user defined functions like **FNVALID**.

Parentheses can be used to ensure that everything within the parentheses is evaluated before any other calculations take place. Indirection operators are described in chapter 39.

Group 2

Raise to the power, eg

```
3^2=9
3^3=27
```

Group 3 and Group 4

These contain all the usual arithmetic operators. Nothing unexpected here.

Group 5

This contains the relational operators which mean 'greater than', 'less than', etc. They are used in expressions such as

```
IF X>10 THEN...
```

Group 6

Logical **AND** is used to ensure that two or more conditions hold true before some action is taken, eg

```
IF X>10 AND Y=6 THEN...
```

For further details see under **AND** in the chapter on BASIC keywords.

Group 7

Logical **OR** is also used with multiple conditions, eg

```
IF X>10 OR Y=6 THEN
```

The action is taken if one or more of the conditions is true. **EOR** is normally only used as a bitwise operator and the user is referred to the BASIC keywords chapter for details.

27 Error handling

If the computer is unable to deal with a situation such as this:

```
PRINT 3/0
```

then it will report the fact to you with an 'error message' and then stop, waiting for your next command

```
>PRINT 3/0
```

```
Division by zero
```

If you are just playing at the keyboard this entry is not a problem – in fact one of the main virtues of BASIC is that it does try to give you an indication of why it is unable to proceed. However if you are writing a program for someone else to use, and you do not want them to be bothered with error messages then you must take the precautions to deal with every possible error that might arise.

The major tool in error handling is the statement

```
ON ERROR GOTO 5000
```

(The 5000 is an example – it could **GOTO** any line number you like.)

Once the computer has encountered an **ON ERROR GOTO** statement it will no longer report errors and stop – instead it will go to line 5000 (or wherever you have told it to go to). The statement **ON ERROR OFF** makes the computer handle errors normally again. The computer has an *error number* for every error it may encounter and you can use the error number to enable you to know what has gone wrong. The error number is stored in the variable **ERR**. The error number for an attempt to divide by zero is 18 for example.

```
  10 ON ERROR GOTO 2000
  20 PRINT "HELLO"
  30 PRINT 3/0
  40 PRINT "BYE"
  50 END
2000 PRINT ERR
>RUN
HELLO
      18
```

The computer also remembers the line at which it detected the error and this number is stored in the variable **ERL**.

```
10 ON ERROR GOTO 2000
20 PRINT "HELLO"
30 PRINT 3/0
40 PRINT "BYE"
50 END
2000 PRINT ERR
2010 PRINT ERL
>RUN
HELLO
        18
        30
```

As you will see from the above the computer detected error number 18 in line number 30. Instead of just printing an error number the computer can be made to deal with the problem. Look at the next program which will generate an error when X gets to zero.

```
100 X=-5
110 PRINT X, 3/X
120 X=X+1
130 IF X<5 THEN GOTO 110
140 END
>RUN
        -5        -0.6
        -4        0.75
        -3         -1
        -2        -1.5
        -1         -3
         0

Division by zero at line 110
```

If we put in error handling routine we can let the computer deal with the problem itself.

```
10 ON ERROR GOTO 1000
100 X=-5
110 PRINT X, 3/X
120 X=X+1
130 IF X<5 THEN GOTO 110
140 END
1000 IF ERR=18 THEN PRINT: GOTO 120
1010 REPORT
>RUN
        -5        -0.6
        -4        -0.75
        -3         -1
```

```
-2      -1.5
-1        -3
 0
 1         3
 2       1.5
 3         1
 4      0.75
```

In the example program above error 18 was dealt with successfully but line 1010 causes it to **REPORT** other errors in the normal way without trying to deal with them.

It is usually easy, but tedious, to anticipate all the likely errors but careful planning is needed if all the error handling is to be effective. In particular you should be aware that when an error occurs you cannot return into a **FOR...NEXT** or **REPEAT...UNTIL** loop or into a procedure or function or subroutine. So long as you are aware of these limitations you can program around them.

28 Teletext control codes and MODE 7

MODE 7 is a Teletext compatible display mode which is very economical in its use of memory. It can provide a full colour text display with limited, but full colour, graphics. This mode is strongly recommended for applications which do not require very fine graphic detail.

MODE 7 uses the standard Teletext control codes to change colours rather than the BBC BASIC **COLOUR** and **DRAW** statements. It cannot be overemphasied that **MODE 7** requires different codes and statements from those available in **MODES 0** to **6**. The **MODE 7** display consists of 20 lines of 40 characters. Each line will normally consist of white letters and numbers (text) on a black background. If the user wishes to change the colour of the text or the background then a control code must be sent to the screen with a **PRINT** or **VDU** statement. By way of explanation let's examine one typical line of characters on a **MODE 7** screen.

The screen display consists of 25 lines (numbered 0 to 24) each containing up to 40 characters (0 to 39).
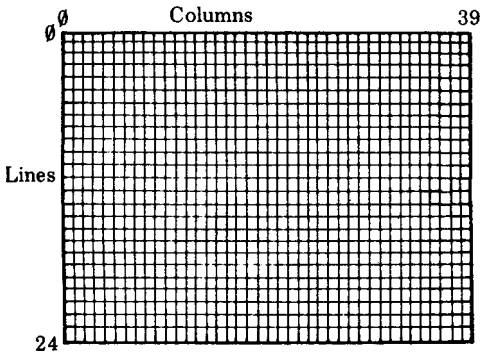
Let's examine a typical line and see what is displayed on the screen.
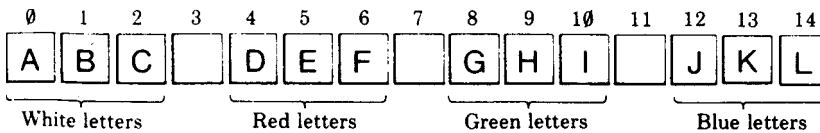
To get this on the screen you will need to type

```
>MODE 7
>PRINT
"ABC";CHR$(129);"DEF";CHR$(130);"GHI";CHR$(132);"JKL"
```

You will see that there is a space on the screen between the letters **ABC** and the letters **DEF**. That space is in fact occupied by an invisible 'control code'. The control code 'appears' on the screen as a spce but it affects everything to its right. The control code at position 3 is code number **129** and that has the effect of turning all letters and numbers that follow into red. The code at position 7 is number **130** which produces green alphanumerics (letters and numbers).

Here is a list of a few more control codes

129	Alphanumeric red
130	Alphanumeric green
131	Alphanumeric yellow
132	Alphanumeric blue
133	Alphanumeric magenta
134	Alphanumeric cyan
135	Alphanumeric white

To change the colour of the text

To get these codes on the screen you have to type **PRINT CHR$(X)** just before the text you want to alter, where **X** stands for the code you want. Often you will wish to put the codes in between two words and you do that by placing both the words and the **CHR$(X)** in one long **PRINT** statement as shown below

```
10 MODE 7
20 PRINT "WHITE";CHR$(131);"YELLOW";CHR$(135);"AND
BACK TO WHITE"
```

To make characters flash

Another code, **136**, makes everything that follows on that line flash. Try

```
10 MODE 7
20 PRINT "HELLO";CHR$(136);"FLASHER!"
```

It must be emphasised that every line starts off in white, non-flashing, normal height, black background and so on. If you want to print a whole page in red letters then every line must start with control code **129**.

You should by now understand how to put the control codes into a **PRINT** statement and now we can see what other effects are available.

To change the background colour requires three control codes. Suppose that you want blue letters on a yellow background then you must use the following sequence of control codes

131 Yellow alphanumeric

157 New background

132 Blue alphanumeric

```
10 MODE 7
20 PRINT CHR$(131);CHR$(157);CHR$(132);"BLUE
LETTERS ON YELLOW"
```

As you will gather to change the background colour you must first select letters of the desired colour, then declare a new background and then reselect the colour of the letter. Note that you cannot have a flashing background, so the following sequence

136 Flash

131 Yellow alphanumeric

157 New background

132 Blue alphanumeric

will produce flashing blue letters on a steady yellow background.

```
10 MODE 7
20 PRINT
CHR$(136);CHR$(131);CHR$(157);CHR$(132);"BLUE
LETTERS ON YELLOW"
```

Flash is turned off with control code **137**.

To produce double height characters

It is possible to write characters with double their normal height using control code **141**. Obviously this takes up two of the normal display lines. What is not so obvious is that you must therefore print exactly the same text on two successive lines. Try the following:

```
10 MODE 7
20 PRINT CHR$(141);"THIS IS DOUBLE HEIGHT"
```

As you will see it only produces the top half of the letters. Add line 30 and it works properly.

```
10 MODE 7
20 PRINT CHR$(141);"THIS IS DOUBLE HEIGHT"
30 PRINT CHR$(141);"THIS IS DOUBLE HEIGHT"
```

Of course you can have (for example) double height, flashing red letters on a white background

141 Double height

157 New background

129 Red alphanumeric

136 Flashing

```
10 MODE 7
20 PRINT CHR$(141);CHR$(157);CHR$(129);CHR$(136);
"THE LOT"
30 PRINT CHR$(141);CHR$(157);CHR$(129);CHR$(136);
"THE LOT"
```

Double height is turned off with control code **140**.

As you can see, it can be very tedious typing in **CHR$(129)** every time you want a Teletext control code. To make things easier it is possible to use the red user defined function keys in combination with the **SHIFT** key to generate these special codes. While pressing **SHIFT** the function keys normally produce the codes shown in the following table.

f0	**128**	No effect	
f1	**129**	Alphanumeric	red
f2	**130**	Alphanumeric	green
f3	**131**	Alphanumeric	yellow
f4	**132**	Alphanumeric	blue
f5	**133**	Alphanumeric	magenta
f6	**134**	Alphanumeric	cyan
f7	**135**	Alphanumeric	white
f8	**136**	Alphanumeric	flash on
f9	**137**	Alphanumeric	flash off

As you will see **f0** is set to produce code **128** and the other keys produce higher numbers. **128** is said to be the 'base address' for the keys. The base address can be altered with ***FX 226** if you wish. See chapter 43 for more details.

Once you have a Teletext control code on the screen you can use the editing keys (eg **COPY**) to copy it into another string. This can be very useful.

Graphics

In addition to displaying coloured letters it is possible in `MODE 7` to do a certain amount of work with graphics. The graphics available in Teletext mode are more complicated to use than in other modes but with a little patience very good effects can be achieved. An additional set of control codes are used to change lower case letters into small graphic shapes. The shapes are all based on a two by three grid, the same total size as a large letter.
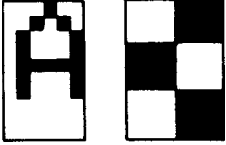
If you want to use those graphic shapes instead of lower case letters then they must be preceded with one of the following control codes:

145   Red graphics

146   Green graphics

147   Yellow graphics

148   Blue graphics

149   Magenta graphics

150   Cyan graphics

151   White graphics

Note that upper case letters will still show as letters in the same colour that you have selected for the graphics. Thus

```
10 PRINT CHR$(145);"ABCdefGHIjkl"
```

will show the following on the screen in red.
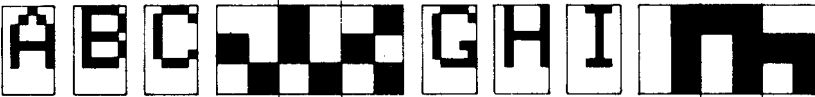
(The full list of graphics shapes is given in Appendix B.)

Graphics codes

It is possible to calculate the code for any particular graphics shape in the following way. Each of the six cells is represented by a specific code number:

In addition you should add in 32 + 128 (ie 160). For example the ASCII code for
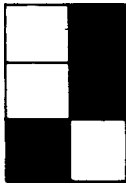
is 2 + 8 + 16 + 32 + 128 = 186.

Making a large shape

In the next chapter you can see how to use user defined characters to draw a space ship. By way of comparison, a similar but cruder space ship can be made in Teletext mode. Here is the design and the code number for each graphic character:
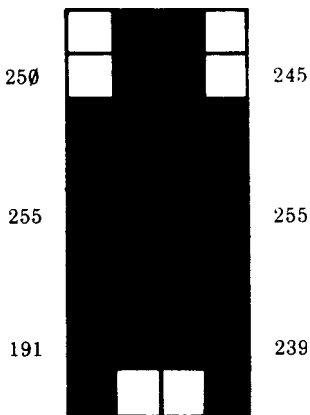
To make these display as graphics characters each line must be preceded by (for example) code **146** (green graphics). So the following codes must be printed on the screen:

```
146, 250, 245
146, 255, 255
146, 191, 239
```

These codes can be sent using **PRINT CHR$( )** as long as you are careful to get each code in the correct place, eg

```
10 MODE 7
20 X=20
30 Y=10
40 PRINT
TAB(X,Y);CHR$(146);CHR$(154);CHR$(250);CHR$(245)
50 PRINT
TAB(X,Y+1);CHR$(146);CHR$(154);CHR$(255);CHR$(255)
60 PRINT
TAB(X,Y+2);CHR$(146);CHR$(154);CHR$(191);CHR$(239)
```

Instead of using **PRINT TAB (X,Y)** it is probably easier in this case to use ASCII codes to move the cursor-down one line (code **10**) and back four spaces (code **8** four times). It is probably also easier to use the **VDU** statement rather than **PRINT CHR$ ( )**. If these two things are done then the program becomes

```
 5 MODE 7
 6 PRINT TAB(20,10);
10 VDU 146,154,250,245
20 VDU 10,8,8,8,8
30 VDU 146,154,255,255
40 VDU 10,8,8,8,8
50 VDU 146,154,191,239
100 PRINT
```

A complete list of the Teletext codes is given in Appemdix A and Appendix B.

Teletext graphics codes for the more adventurous

As you will have realised, the statements **MOVE** and **DRAW** are not available in the Teletext mode(**MODE 7**). If you wish to draw lines in this mode you will need to use a suitable procedure – so here is one. It uses a look-up table, called **S%**, to remember the numbers corresponding to each of the six pixels (picture elements) in a Teletext graphics character. The look-up table must be set up at the beginning of the program:

```
10 DIM S% 7
20 !S%=&08040201
30 S%!4=&4010
```

(Refer to chapter 39 for an explanation of the above techniques).

Next a row of Teletext control codes must be written down the left hand side of the screen to convert every line into a graphics display:

```
40 PROCGR
```

and the associated procedure is:

```
200 DEF PROCGR
210 LOCAL Y%
220 VDU 12
230 FOR Y%=0 TO 18
240 VDU 10,13,&97
250 NEXT
260 ENDPROC
```

The main program – in this case to plot a 'sine curve' – follows:

```
50 FOR X=0 TO 75 STEP 0.25
60 PROCPLOT (X,28+28*SIN(X/10))
70 NEXT
80 END
```

and lastly here is the procedure to plot the point:

```
300 DEF PROCPLOT(X%,Y%)
310 LOCAL C%,A%
330 VDU 31,X% DIV2+1, 19-Y% DIV3
340 C%=S%?((X% AND 1)+(2-Y%MOD3)*2)
350 A%=135
360 VDU (USR &FFF4 AND &FF00) DIV256 OR C% OR 128
370 ENDPROC
```

There is no need to know how it works but here is an explanation in case you are interested.

The X and Y coordinates are put into **X%** and **Y%** and then line 330 moves the text cursor to the position **X%, Y%**.

Line 340 uses the look-up table (**S%**) to calculate the 'value' (in terms of ASCII code) of the selected pixel at **X%, Y%**.

Setting **A%=135** and jumping to the subroutine at **&FFF4** returns the ASCII code of the character which includes the spot **X%, Y%**. See chapter 43, which explains this OSBYTE call for a similar example. The character read from the screen is then ORed with the new pixel (**C%**) and written with the **VDU** statement.

The `&97` in line 240 produces white graphic dots. Other values will give other colours. For example `&91` would draw a red graph.

Two improvements could be made; first we could test for illegal values of `X%` and `Y%` with

```
320 IF X%<0 OR X%>75 OR Y%<0 OR Y%>56 THEN ENDPROC
```

and secondly remember the position of the cursor before we entered the procedure and restore the cursor to that position at the end of the procedure.

Notice that you do not need to reference actual memory coordinates and this is vital if your programs are to work via the Tube. You may not think it is important now but you will find that it is advisable to write programs using the machine code calls provided and not to get into the habit of addressing the memory directly.

29 Advanced graphics

As we saw earlier, the following keywords can be used in a variety of statements which produce high resolution graphics effects on the screen:

MODE   Selects a particular graphics **MODE**

GCOL   Selects the colour and drawing 'style' of any graphics (except in **MODES 3**, **6**, or **7**)

DRAW   Draws lines (except in **MODES 3**, 6, or **7**)

MOVE   Moves the graphics cursor (except in **MODES 3**, **6**, or **7**)

PLOT   Draws lines, dotted lines, points and colours in triangles (except in **MODES 3**, **6**, or **7**)

These will only become familiar with use. What follows is a description of how to use some of the keywords to produce a selection of results on the screen.

How to change the screen display modes

The screen mode can be changed at any time by typing **MODE** X, where X is the value 0 to 7 from the following list:

MODE 0   Uses two colours with very high resolution and requires 16K of memory to 'map' the screen

MODE 1   Uses four colours with high resolution and requires 16K of memory

MODE 2   Uses four colours with medium resolution graphics – 16K

MODE 3   Text only – 16K

MODE 4   Two colours and high resolution graphics – 8K

MODE 5   Four colours and medium resolution graphics – 8K

MODE 6   Text only – 8K

MODE 7   Teletext (which is the subject of a separate chapter of this book)– 8K

In **MODES 0**, **1**, **2**, **4** and **5** the screen is divided up into imaginary rectangles, like a piece of graph paper. In **MODE 0** there are 640x256 squares; in **MODE**s **1** and **4** there are 320x256 and in **MODE**s **2** and **5** there are 160x256 (in other words, the higher the resolution of the graphics, the smaller the rectangle). The higher the resolution , the more memory is used up in the process of 'mapping' the screen.

MODES 128-135 do not use any of main memory, and are the 'shadow screen' equivalents of **MODES 0-7**. See chapter 42 for more details.

How to draw lines

In Appendix E you will find a graphics planning sheet which shows the way the screen can be thought of as a piece of graph paper, with each point having a horizontal (X) and a vertical (Y) value. The 'origin' is point 0,0 and is at the bottom left of the screen. Top right is 1279,1023.

How to draw a square in the centre of the screen

1. First set up a screen mode which can support graphics. Use the **MOVE** statement to move the graphics 'cursor' from its home position (0,0) to a point where we can start drawing (say 400 units along and 400 units up).

```
100 MODE 5
110 MOVE 400,400
```

2. Draw a line horizontally to point 800,400. The **DRAW** command draws a line from the last point 'plotted' to a point defined in the **DRAW** statement.

```
120 DRAW 800,400
```

3. Finish the box with three more **DRAW** statements

```
130 DRAW 800,800
140 DRAW 400,800
150 DRAW 400,400
```

 Run the program.

Changing the colour of the square

The normal colour in **MODE 5** is white. Add the following line:

```
105 GCOL 0,1
```

This changes the lines to 'logical' colour 1, which in **MODE 5** is red. **MODE 5** only has four colours. When the machine is switched on they are black, red, yellow and white. However, logical colours can be changed by using one of the VDU commands (see later). So, if you know what you are doing, you can select any four colours in **MODE5**.

How to fill in with colour

PLOT 85,X,Y (see the BASIC keywords chapter) draws and then fills in a triangle drawn from the last two plotted points to the point defined by X and Y. The colour is the current graphics foreground colour. Add the following line:

```
160 PLOT 85,800,800
```

Run the program.

This will fill in one half of the square.

Now add:

```
170 PLOT 85,800,400
```

Run the program and the whole square should become red.

How to change colours

At any particular moment the computer can print and draw on the screen using four colours. This page uses two 'colours': a white background colour and a black foreground colour – that is, black writing on a white background. Similarly, the computer has a text background colour and a text foreground colour but, in addition, it is aware of a graphics foreground colour (used to draw lines), and a graphics background colour. When you change **MODE** the computer resets all these colours as follows:

– Text foreground colour: white
– Text background colour: black
– Graphics foreground colour: white
– Graphics background colour: black

The number of colours that can appear on the screen at one time depends on the **MODE** selected. In **MODES 0, 3, 4** and **6** you can only have two colours at any time and they are normally black and white. In **MODES 1** and **5** you can have up to four colours at any time and they are normally black, red, yellow and white. In **MODE 2** you can have up to 16 different coloured effects.

Let us consider **MODE 5** for a moment and explore the effects that are available. **MODE 5** is a four colour mode and the default colours are black, red, yellow and white. As you may have gathered, text and graphics are dealt with separately so to change the colour that will be used for text output type

COLOUR 0 To give black text

COLOUR 1 To give red text

COLOUR 2 To give yellow text

COLOUR 3 To give white text

However, to change the colour used for graphics, for example to produce lines with the **DRAW** statement, you use these statements

GCOL 0,0 Black graphics

GCOL 0,1 Red graphics

GCOL 0,2 Yellow graphics

GCOL 0,3 White graphics

The two groups of statements above change the 'text foreground' and 'graphics foreground' colours.

You will have noticed that, so far, 1 represents red, 2 is yellow and so on. To change the background colours we add 128 to these numbers. Thus **COLOUR 129** will give a red text background, and **GCOL 0,129** would set the graphics background colour to red.

For text, for example, to change from white lettering on black, to black lettering on red in **MODE 5,** type

```
MODE 5
COLOUR 129
COLOUR 0
CLS
```

All text will now be in black and red. Graphics will still appear in white.

To change text colours in the middle of a program simply insert the appropriate colour statements before the print statements to which they refer.

For graphics use the **GCOL** statement which stands for 'Graphics COLours'. **GCOL** has two numbers after it (see the BASIC keywords chapter). The second number refers to the logical colour which is to be used for graphics in the future. The first number is usually set at 0.

So for example, to get red graphics lines on a yellow background, type

```
COLOUR 131
GCOL 0,1
CLS
```

But suppose that you wanted a blue background in **MODE 5**. So far the only available colours have been black, red, yellow and white and there is a limit of four colours in **MODE 5**. You can make one of four colours blue if you want to. You do this with the statement

```
VDU 19,0,4,0,0,0
```

and then the four available colours would be blue, red, yellow and white.

In **MODE 5**, only four colours are available at a time and they are referred to as 'logical' colours 0 to 3. In **MODE 5** 'logical' colour 0 is normally black, 'logical' colour 1 is normally red and so on but you can change the 'actual colour' of the 'logical colours' easily by using the **VDU 19** statement followed by five numbers, separated by commas.

In a two colour mode such as **MODE 4** we can do similar things eg

```
MODE 4
```

```
VDU 19,1,2,0,0,0
```

changes logical foreground colour 1 (which is initially white) to actual colour 2 which is green, and

```
VDU19,0,5,0,0,0
```

changes logical background colour 0 to actual colour 5, which is magenta. (*Note:* The zero at the end are for future expansion of the system.) The computer will now produce these colours until either it is switched off, the **BREAK** button is pressed, or the **MODE** is changed. These instructions to change the colours can be embedded in a program thus making it possible to alter the colours while a program is running.

Here is a list of the numbers for each 'actual colour' that the computer can produce.

Actual colour number	Displayed colour
0	Black
1	Red
2	Green
3	Yellow
4	Blue
5	Magenta
6	Cyan
7	White
8	Flashing black/white
9	Flashing red/cyan
10	Flashing green/magenta
11	Flashing yellow/blue
12	Flashing blue/yellow
13	Flashing magenta/green
14	Flashing cyan/red
15	Flashing white/black

So 'actual colour' numbers are any numbers between 0 and 15. To make logical 3 a flashing red/cyan effect you write

```
VDU 19,3,9,0,0,0
```

and here are some other examples

```
VDU 19,1,2,0,0,0
```   Logical colour is green

```
VDU 19,3,5,0,0,0
```   Logical colour 3 is magenta

```
VDU 19,0,12,0,0,0
``` Logical colour 0 is flashing blue/yellow

Having set the logical colours up in this way you could then select logical colour 0 (flashing blue/yellow) as the text foreground colour whith `COLOUR 0`, or as the graphics foreground colour with `GCOL 0,0`. The table shows the set up for each time you change `MODE`.

| Foreground colour | | Background colour | |
|---|---|---|---|
| **Logical number** | **Actual colour** | **Logical number** | **Actual colour** |
| **Modes 0,3,4,6** | | | |
| 0 | Black (0) | 128 | Black (0) |
| 1 | White (7) | 129 | White (7) |
| **Modes 1,5** | | | |
| 0 | Black (0) | 128 | Black (0) |
| 1 | Red (1) | 129 | Red (1) |
| 2 | Yellow (3) | 130 | Yellow (3) |
| 3 | White (7) | 131 | White (7) |
| **Mode 2** | | | |
| 0 | Black (0) | 128 | Black (0) |
| 1 | Red (1) | 129 | Red (1) |
| 2 | Green (2) | 130 | Green (2) |
| 3 | Yellow (3) | 131 | Yellow (3) |
| 4 | Blue (4) | 132 | Blue (4) |
| 5 | Magenta (5) | 133 | Magenta (5) |
| 6 | Cyan (6) | 134 | Cyan (6) |
| 7 | White (7) | 135 | White (7) |
| 8 | Flashing black/white (8) | 136 | Flashing black/white (8) |
| 9 | Flashing red/cyan (9) | 137 | Flashing red/cyan (9) |
| 10 | Flashing green/magenta (10) | 138 | Flashing green/magenta (10) |
| 11 | Flashing yellow/blue (11) | 139 | Flashing yellow/blue (11) |
| 12 | Flashing blue/yellow (12) | 140 | Flashing blue/yellow (12) |
| 13 | Flashing magenta/green (13) | 141 | Flashing magenta/green (13) |
| 14 | Flashing cyan/red (14) | 142 | Flashing cyan/red (14) |
| 15 | Flashing white/black (15) | 143 | Flashing white/black (15) |

You should note that the `GCOL` statement is followed by two numbers.

The first number can be used to control the way that the colour (which is selected by the second number) is affected by what is already on the screen. The statement

```
GCOL 0,3
```

tells the computer that the graphics colour to be used is to be logical colour 3 and that this is to appear no matter what was on the screen under the new line or triangle. Values other than 0, for the first number, have other effects. For example, a value of 4, as in **GCOL 4,0** has the effect of drawing a line which is the 'inverse' logical colour to the colour that it is currently crossing over.

In two colour **MODE**s the inverse of logical colour 0 is logical colour 1. In four colour modes the following applies.

Logical colour	Inverse
0	3
1	2
2	1
3	0

With 'default' actual colour of black, red, yellow and white the inverse colours would be white, yellow, red and black.

In **MODE 2**, the 16 colour **MODE**, all steady colours translate to flashing colours and vice versa as the next table shows for the default colours.

Logical colour	Default displayed colour	Inverse
0	Black	Flashing white/black
1	Red	Flashing cyan/red
2	Green	Flashing magenta/green
3	Yellow	Flashing blue/yellow
4	Blue	Flashing yellow/blue
5	Magenta	Flashing green/magenta
6	Cyan	Flashing red/cyan
7	White	Flashing black/white
8	Flashing black/white	White
9	Flashing red/cyan	Cyan
10	Flashing green/magenta	Magenta
11	Flashing yellow/blue	Blue
12	Flashing blue/yellow	Yellow
13	Flashing magenta/green	Green
14	Flashing cyan/red	Red
15	Flashing white/black	Black

Other values of the first number following GCOL enable the logical colour to be plotted to be ANDed, ORed or Exclusive-ORed with the logical colour presently on the screen. The user is referred to the BASIC keywords chapter for a description of AND, OR and EOR but the following examples may help.

In MODE 5 with the background set to red by the following statements

```
MODE 5
GCOL 0,129
CLG
```

an attempt to draw a yellow line ORed with the background colour by using the statement

```
GCOL 1,2
```

would in fact produce a white line since the background logical colour is 1 and the new line is to be drawn in logical colour (1 OR 2) ie logical colour 3. The same principles apply to GCOL 2, which ANDs the new colour with the previously displayed logical colour and to GCOL 3, which Exclusive-ORs the colours together.

Obviously, an understanding of AND, OR and EOR is required before all the GCOL statements can be used. The effects which can be produced are very useful when it comes to producing sophisticated animations.

How to plot a point on the screen

The PLOT command can also be used to plot points.

Type MODE 4 (which is a two colour MODE) and type

```
PLOT 69,500,500
```

This will plot a point in white at coordinates X=500, Y=500. Type

```
PLOT 69,600,500
```

and another point will appear at X=600, Y=500.

How to remove a point selectively

Typing CLG would clear the whole graphics area. However, instead of CLG , type

```
PLOT 70,500,500
```

and you will see that one of the points has gone. This is because PLOT 70 prints the 'inverse' colour at the given point. This is particularly useful when 'animating', for example, a point (see below). In MODE 4 the 'inverse' of logical colour 1 is logical colour 0.

Suppose we want, for example, a yellow dot on a blue background. We need to change the foreground colour from white to yellow and the background colour from black to blue. To do this, use the `VDU 19` command as described earlier:

```
VDU 19,1,3,0,0,0
```

This alters logical colour 1 (this means the normal foreground colour) to actual colour 3 (ie yellow). Now type

```
VDU 19,0,4,0,0,0
```

This alters logical colour 0 (which is the normal background colour) to actual colour 4 (ie blue).

```
PLOT 69, 500, 500
```

will now produce the desired effect.

Animation

How to make a ball and move it on the screen

This time we'll write a program in steps to make a yellow 'ball' consisting of four dots which move on the screen on a red background.

1. Setting up `MODE` and colours

```
10 MODE 4
20 VDU 19,1,3,0,0,0
30 VDU 19,0,1,0,0,0
```

2. Next a 'procedure' for creating a ball. This procedure will be 'called up' whenever we want to create a ball on the screen at a point (X,Y). It is good practice to put a procedure like this at the end of a program, so we'll give it a high line number. X and Y are the parameters for this procedure.

```
1000 DEF PROCBALL (X,Y)
1010 PLOT 70,X,Y
1020 PLOT 70,X,Y+4
1030 PLOT 70,X+4,Y
1040 PLOT 70,X+4,Y+4
1050 ENDPROC
```

We use `PLOT 70` rather than `PLOT 69` to help the animation which follows.

3. Making the ball travel horizontally at height Y = 500.

```
40 REM HORIZONTAL MOVEMENT
50 FOR N = 1 TO 1000
60 PROCBALL (N,500)
70 PROCBALL (N,500)
```

```
80 NEXT N
90 END
```

You will see that this prints the ball at the point (N,500) and then 'unprints' it only to print it again one step further on, and so on.

To speed the ball up, alter line 50:

```
50 FOR N = 1 TO 1000 STEP 10
```

How to create your own 'graphics' characters

Each character which you type in at the keyboard has an associated ASCII code. When the computer is told to print this character it looks up the code and prints the appropriate character as an eight by eight matrix of dots. The letter 'A', for example, has the code value 65 and 'a' has the value 97. (See Appendix A for the other codes.) However, certain code values have been left to be defined by the user. They include values 224 to 255. (See chapter 34 for more details.) They can be defined by use of the **VDU 23** command.

How to make a character (eg a man)

Create the character by planning it on an eight by eight square grid.

Note the numbers along the top of the grid which start at the right and double at each column to the left.

To store the character shown above as code number 240, type in

```
VDU 23,240,28,28,8,127,8,20,34,65
```

The numbers which follow **VDU 23,240** tell the computer the pattern of dots in each horizontal row. These values are the 'byte' patterns corresponding to the eight cells of each row. They can be calculated in a number of ways and entered as a decimal or hexadecimal number. The simplest way for the novice is to add up the values shown in the diagram above. Thus row a consists of 16+8+4=28, row b is also 28, the third row is 8, the fourth row is 64+32+16+8+4+2+1=127, and so on. So to create the little man, type the following program:

```
 5 MODE 5
```

```
10 VDU 23,240,28,28,8,127,8,20,34,65
20 PRINT CHR$(240);
30 GOTO 20
```

Note the last two lines, which print him over and over again.

How to make him move

We have created a character which can be reproduced in any **MODE** (except **MODE 7**). By printing him and then erasing him at successive positions he can be made to move across the screen in a similar way to the ball. However, since he exists as a character he is treated as text, not graphics. This means that he is made to appear by using **PRINT** – as above – and the position can be defined by using the **TAB** statement.

Try this:

```
5 MODE 4
10 VDU 23,240,28,28,8,127,8,20,34,65
20 PRINT TAB(20,10); CHR$(240)
```

The character appears at text position 20,10. This is a 40 character wide **MODE** so 20,10 is roughly in the middle of the screen.

Now try this:

```
20 FOR X = 1 TO 19
30 PRINT TAB(X,10); CHR$(240)
40 NEXT X
```

This will print the man 19 times across the screen. Now type these additional lines:

```
40 FOR T = 1 TO 100: NEXT T
50 PRINT TAB(X,10) ; ""
60 NEXT X
```

and he appears to run across the screen.

By alternating lines 30 and 50 so that the value in parentheses is (X,X) he can be made to move diagonally across the screen.

Note that line 40 acts as a time delay.

How to make a larger character

This time, a space ship.

Plan this by using several grids.

So we have

```
  5 MODE 4
 10 VDU 23,240,8,8,28,28,62,62,62,62
 20 VDU 23,241,62,62,62,62,62,62,62,62
 30 VDU 23,242,62,62,62,127,127,127,93,93
 40 X = 20: Y = 10
 50 PRINT TAB(X,Y);CHR$240;
 60 PRINT TAB(X,Y+1);CHR$241;
 70 PRINT TAB(X,Y+2);CHR$242;
```

This produces the space ship in the middle of the screen. To make it take off, change the program by adding these lines:

```
  7 VDU23,1,0;0;0;0;
 35 X=20
 40 FOR Y = 24 TO 0 STEP -1
 80 FOR T = 1 TO 100: NEXT T
 90 PRINT TAB(X,Y); " ";
100 PRINT TAB(X,Y+1); " ";
110 PRINT TAB(X,Y+2); " ";
120 NEXT Y
```

Now add an extra character to produce flames at the bottom of the space ship for the initial take-off.

```
 32 VDU 23,243,28,60,30,60,126,108,162,162
 75 IF Y>12 THEN PRINT TAB(X,Y+3);CHR$243;
115 PRINT TAB(X,Y+3); " ";
```

How to make the movement smoother

The rocket does not appear to move up the screen smoothly but in a series of jumps. This is because when you use **TAB(X,Y)** there are only 32 possible lines you can print on.

The **VDU5** statement (see also chapter 10, where it is used for positioning accents, etc) causes text to be written at the graphics cursor. This means that you can move to any point on the screen on the normal 1280×1024 graphics 'grid' and print text or user defined characters. To do this **MOVE X,Y** is used.

VDU 4 undoes the effect of the **VDU 5** statement. It causes text to be written where the text cursor is.

Then the first character can be printed using **PRINT CHR$(240);** or **VDU 240**. These two alternatives are equivalent but **VDU 240** means less typing! We now need to backspace one character to our original position and move down one character cell so we can print the next character. We can't use **TAB(X,Y)** because it won't work after a **VDU 5** statement, so we use two of the four 'cursor movement commands' (see chapter 10).

VDU 8 Backspace cursor one character

VDU 9 Forwardspace cursor one character

VDU 10 Move cursor down one line

VDU 11 Move cursor up one line

Later on we'll also use **VDU 127** which has exactly the same effect as pressing the **DELETE** key. All the **VDU** commands are listed at the back of the book – there's no need to remember them all.

We need to use **VDU 8** and then **VDU 10**, or the other way around. You can string **VDU** commands together, so we can use

VDU 240,8,10

which will print the first character and then move the cursor into position to print the next one. To print the whole rocket and flames this is repeated three times: here it is done in **PROCROCKET**.

```
 5 MODE 4
 7 VDU 23,1,0;0;0;0;:REM turn off cursor
10 VDU 23,240,8,8,28,28,62,62,62,62
20 VDU 23,241,62,62,62,62,62,62,62,62
30 VDU 23,242,62,62,62,127,127,127,93,93
```

```
  32 VDU 23,243,28,60,30,60,126,108,162,162
  35 X%=600
  37 VDU 5
  38 GCOL 4,1
  40 FOR Y%=120 TO 1023 STEP 10
  50 PROCROCKET: REM draw the rocket
  90 PROCROCKET: REM now delete it
 120 NEXT Y%
 130 END
1010 DEF PROCROCKET
1020 MOVE X%,Y%
1025 VDU 240
1030 VDU 10,8,241
1040 VDU 10,8,242
1050 IF Y%<500 THEN VDU 10,8,243
1060 ENDPROC
```

Notice that 'integer' variables (followed by a % sign) are used: these make the program run considerably faster than 'real' variables.

The rocket is printed at line 50 by **PROCROCKET**. Line 90 prints the rocket again in its inverse colour (as specified in line 38 **GCOL 4,1**), and so deletes it.

As an alternative to this, the rocket could be deleted by replacing line 90 with **PROCdelete**, deleting line 38, and adding the new procedure

```
2000 DEF PROCdelete
2010 MOVE X%,Y%
2020 VDU 9,127
2030 VDU 10,9,127
2040 VDU 10,9,127
2050 VDU 10,9,127
2060 ENDPROC
```

This just deletes the rocket with **VDU 127**.

We can get smoother movement if we just delete the bottom character every time. This removes most of the flicker and what remains becomes a good effect. The price is that the detail at the bottom of the rocket is lost, so this method only works if your character gets wider at the bottom!

Delete line 90, and type this new procedure:

```
  38 GCOL,1
  40 FOR Y%=120 TO 1023 STEP 4
1010 DEF PROCROCKET
1020 MOVE X%,Y%
1025 VDU 240,10,8
1030 VDU 241,10,8
```

```
1040 VDU 242
1050 VDU 10,8,243
1055 VDU 127
1060 ENDPROC
```

If you change one line

```
1050 IF Y%<500 THEN VDU 10,8,243 ELSE VDU 10
```

the flames will cut out after take-off again.

Making a complete lunar landing game

Using the procedures we have developed for creating and moving the rocket on the screen, we can incorporate these into a complete game which can test your skill at landing a space ship on the moon. This complete program uses a range of techniques described elsewhere in the book and was written by Jim Murray. Included are a few notes to explain what is going on.

```
  5 ON ERROR REPORT:GOTO 245
 10 MODE5
 20 VDU 23,240,8,8,28,28,62,62,62,62
 30 VDU 23,241,62,62,62,62,62,62,62,62,
 40 VDU 23,242,62,62,62,127,127,127,93,93
 50 VDU 23,243,28,60,30,60,126,108,162,162
 60 VDU 23,1,0;0;0;0;
 70 VDU 19,2,2,0,0,0
 80 VDU 28,0,20,14,0
 90 @%=&906
110 *FX 11,1
```

Notes:
Line 70 enables us to use green.
Line 80 sets up a text window.
Line 90 sets `@%` – so numbers are printed as we want them.
Line 110 sets the auto-repeat delay period to its minimum value.
Line 5 disables the effect of line 110 if you press **ESCAPE**, otherwise it's difficult to type anything again!

```
120 PROClabels
130 PROCmoon
140 PROCinitialise
150 VDU 5
160 X%=960
165 GCOL 0,3
170 REPEAT
180 burn$=INKEY$(0)
185 *FX 15,1
```

```
190 IF burn$=""THEN burnrate%=0 ELSE
burnrate%=VAL(burn$)*30
200 PROCcalculate
210 PROCdashboard
220 IF Y%>oldY%+4 OR Y%<oldY%-4 THEN PROCrocket
225 PROCburn
230 UNTIL height=0
240 IF speed>0.004 THEN PROCcrash ELSE PROCfanfare
245 *FX 12,0
247 *FX 15,1
250 END
```

Notes: Although line 250 says **END** this is not the end of the program. What follows are the various procedures which have been called by the program as it exists so far.

Before we give the procedures, some notes on the earlier lines.

Line 120 **PROClabels** – sets up the titles
Line 130 **PROCmoon** – draws moon's surface.
Line 140 **PROCinitialise** – sets all the variables to initial values.
Lines 170 to 230 are the main part of the program – a **REPEAT...UNTIL** loop.
Line 180 – checks to see if any key has been pressed.
Line 185 – clears the key buffer, otherwise the burn continues for a long time after the key is released.
Line 190 – we use **INKEY$** to check if anything has been pressed, but this returns a string. Line 190 converts it to a number.
Line 200 **PROCcalculate** – does the maths.
Line 210 **PROCdashboard** – prints up the results.
Line 220 – prints the rocket if **Y%** (the variable used in **MOVE X%,Y%**) has gone up or down by 4. In this MODE the rocket is printed in the same place unless the change is greater than 4.
Line 225 **PROCburn** – draws the burning fuel.
Line 240 – crash or good landing? – at less than 15 mph it's good.
Line 247 – clears keyboard buffer again so you don't get a string of numbers printed when the program stops.

```
700 DEF PROClabels
710 PRINT TAB(0,7)"secs"
720 PRINT TAB(0,9)"miles"
725 PRINT TAB(0,10)"feet"
730 PRINT TAB(0,12)"speed"
740 PRINT TAB(0,14)"fuel"
750 PRINT TAB(0,16)"burn?"
760 ENDPROC
```

```
 800 DEF PROCmoon
 805 GCOL 0,2
 810 LOCAL X
 820 FOR X=100 TO 1280 STEP 200
 830 MOVE X,0
 840 PLOT 85,X,30
 850 PLOT 85,X+100,0
 860 NEXT X
 870 ENDPROC
 900 DEF PROCinitialise
 910 TIME=0:now=0
 920 speed=1 :REM in miles/second
 930 height=46:REM in miles
 935 Y%=920
 937 oldY%=Y%
 940 gravity=0.001
 950 fuel=16500
 960 totalmass=33000
 965 burnrate%=0
 970 ENDPROC
1100 DEF PROCcalculate
1105 IF fuel<=0 THEN fuel=0:burnrate%=0
1110 burntime=(TIME-now)/100
1120 now=TIME
1130 slower=(burnrate%/
totalmass)*2*EXP(burnrate%*burntime/totalmass)
1140 height=height-speed*burnttime-
burntime*burntime/2*
(gravity-slower)
1150 speed=speed+burntime*(gravity-slower)
1160 burnt=burnrate%*burntime
1170 totalmass=totalmass-burnt
1180 fuel=fuel-burnt
1190 IF height<0 THEN height=0
1200 Y%=height*20+32
1210 ENDPROC

1300 DEF PROCdashboard
1310 VDU4
1320 PRINT TAB(5,7)INT(TIME/100)
1330 PRINT TAB(5,9)INT(height)
1340 PRINT TAB(5,10)INT(height*5280) MOD 5280
1350 PRINT TAB(5,12)INT(speed*3600)
1360 PRINT TAB(5,14)INT(fuel)
```

```
1370 PRINT TAB(5,16)burnrate%
1375 VDU5
1380 ENDPROC

5000 DEF PROCcrash
5020 SOUND 4,-15,100,70
5030 FORX=1 TO 100
5040 MOVE 850+RND(200),RND(200)
5045 GCOL, RND(4)
5050 DRAW RND(1280),RND(1024)
5055 NEXT
5060 ENDPROC

6000 DEF PROCfanfare
6010 FOR X=1 TO 11
6015 READ P,D
6017 IF P=999 THEN L=0 ELSE L=-15
6020 SOUND 1,L,P,D
6025 SOUND 1,0,0,3
6030 NEXT
6035 DATA
97,15,97,5,101,5,999,5,101,5,97,5,101,10,97,2,89,5,81
,5,77,10
6040 ENDPROC
8000 DEFPROCburn
8005 GCOL0,1
8010 MOVEX%,oldY%
8015 IF burnrate%=0 THEN VDU10,9,127 ELSE VDU10,243
8025 GCOL0,3
8030 ENDPROC

10000 DEFPROCrocket
10100 MOVEX%,oldY%
10110 VDU 10,9,127,11,9,127,11,9,127,11,9,127
10120 MOVE X%,Y%
10140 VDU 242,8,11,241,8,11,240
10150 oldY%=Y%
10160 ENDPROC
```

Running the program

You start off at a height of 46 miles moving at 1 mile/sec or 3600 mph. You have fuel of 16500lbs and your weight to start with, including fuel, is 33000 lbs. You fire the rockets by pressing one of the keys 1-9 and holding it down until you want to stop burning. The rate of burning is proportional to the number. You must land at less than 15 mph.

30 Sound

The BBC Microcomputer contains integrated circuits specifically designed to generate musical sounds and noises on four 'channels'. Two statements control the generation of musical sounds; they are **SOUND** and **ENVELOPE** . For simple effects the statement **SOUND** can be used by itself but if the user wishes to have greater control over the quality of the sounds generated then **ENVELOPE** can be used. At its simplest the sound statement is followed by four numbers, eg

```
SOUND C,A,P,D
```

C is the channel number	0 to 3
A is the amplitude or loudness	0 to -15
P is the pitch	0 to 255
D is the duration	1 to 255

The channel number C, determines which of the four 'voices' is to be used. Channel 0 produces 'noise' (this channel will be explained in detail later) whereas channels 1, 2 and 3 produce purer notes.

The amplitude, A, can be varied between 0 (off) and -15(loud).

The pitch, P, selects notes in quarter semi-tone intervals. Middle C is produced when P is set at 52 and other notes are generated with the values of P shown in the table.

As you can see the computer can produce notes spanning five full octaves. The values of P are also shown in the table for a stave in key of C but one octave up.

The duration, D, determines the length of the note and is given in twentieths of a second. Those used to reading music will find that music marked 'Moderato ♩=60' will sound about right with the following settings for D.

			Octave number				
Note	**1**	**2**	**3**	**4**	**5**	**6**	
B	0	48	96	144	192	240	
C	4	*52	100	148	196	244	*middle C
C#	8	56	104	152	200	248	
D	12	60	108	156	204	252	
D#	16	64	112	160	208		
E	20	68	116	164	212		
F	24	72	120	168	216		
F#	28	76	124	172	220		
G	32	80	128	176	224		
G#	36	84	132	180	228		
A	40	88	136	184	232		
A#	44	92	140	188	236		

That completes the simple description of the **SOUND** command.

There are two main areas where the **SOUND** command can be extended. First, instead of working with a fixed sound quality, one can select an 'envelope' to vary both the amplitude and the pitch of the note while it is playing; secondly it is possible to ensure that notes are synchronised so that chords start together. In addition to these major extensions there are a number of other things that can be controlled, and these will be described later.

If you wish to use an envelope to vary either the amplitude or the pitch of a note (or both) then you must first define the envelope and secondly, instead of using a fixed amplitude in the **SOUND** statement, you must quote the envelope number for A. Four envelopes are normally permitted and they are numbered 1 to 4.

Thus

```
SOUND 1,2,53,20
```

would produce on channel 1 a note of middle C with a duration of one second and the amplitude and pitch would be controlled by the envelope number 2.

The statement **ENVELOPE** is followed by 14 numbers and the following labels will be used for the 14 parameters.

```
ENVELOPE N, T, PI1, P12, PI3, PN1, PN2, PN3, AA, AD
,AS, AR, ALA, ALD
```

A brief description of each parameter follows.

Parameter		Range	Function
N		1 to 4	Envelope number
T	bits 0-6	0 to 127	Length of each step in hundredths of a second
	bit 7	0 or 1	0=auto-repeat pitch envelope
			1=don't auto-repeat pitch envelope
PI1		-128 to 127	Change of pitch per step in section 1
PI2		-128 to 127	Change of pitch per step in section 2
PI3		-128 to 127	Change of pitch per step in section 3
PN1		0 to 255	Number of steps in section 1
PN2		0 to 255	Number of steps in section 2
PN3		0 to 255	Number of step in section 3
AA		-127 to 127	Change of amplitude per step during attack phase
AD		-127 to 127	Change of amplitude per step during decay phase
AS		-127 to 0	Change of amplitude per step during sustain phase
AR		-127 to 0	Change of amplitude per step during release phase
ALA		0 to 126	Target level at end of attack phase
ALD		0 to 126	Target level at end of decay phase

Note that the pitch can take on a value between 0 and 255. If the pitch is greater than 255 (eg 257) then 256 will be repeatedly subtracted from it until it is in range.

The amplitude has a range of 0 to 127 in the **ENVELOPE** statement whereas it had a range of 0 to -15 in the **SOUND** statement. The amplitude cannot be set outside the range 0 to 127.

Note also that the total duration of the attack, decay and sustain periods (but not the release period) is determined by the **SOUND** statement and not the **ENVELOPE** statement.

The envelope is divided up into a number of steps – usually a hundredth of a second each and both the pitch and amplitude can be changed at the end of each step.

The pitch envelope

The pitch of the note can be changed in three sections. For each section you can specify the change in pitch at each tick of the clock (step) in the section. Suppose we wish to generate a wailing sound like a police siren. The pitch has to rise and fall like this:

During section 1 the pitch changes +2 units per step and section 1 contains 10 steps. In section 2 the pitch changes -2 units per step and there are 20 steps. Section 3 contains 10 steps of +2 units. So thus far the **ENVELOPE** command looks like

```
ENVELOPE 2,1,2,-2,2,10,20,10
```

The next six numbers control the amplitude of the sound and might well be 1,0,0,-1,100,100 (these will be explained in a moment).

So the total program to show the pitch envelope working would be

```
10 ENVELOPE 2,1,2,-2,2,10,20,10,1,0,0,-1,100,100
20 SOUND 1,2,100,100
```

Here is another pitch envelope – it plays three notes in succession.

```
10 ENVELOPE 3,25,16,12,8,1,1,1,10,-10,0,-10,100,50
```

It reads:

– Envelope number 3.
– Each step is 25/100 ie 1/4 second long.
– The first section of the pitch envelope uses a pitch change of 16 units.
– The second section uses a pitch change of 12 units.
– The third section has a pitch change of 8 units.
– All three sections have only 1 step in each section.
– Now to explain the amplitude envelope.

The amplitude envelope

Suppose that we wish to imitate a car driving towards us getting louder all the time and then driving past before stopping nearby and then driving away. The amplitude of the sound against time might well look like this:

The first phase of the amplitude envelope, where the sound is getting louder is called the 'attack phase'.

The amplitude envelope is specified by giving six parameters. The first (AA) gives the change of amplitude at the end of each step during the attack phase and it must be a positive number. Usually the envelope starts with an amplitude of zero. However it is possible to start with a non-zero amplitude if you have just interrupted a note on the same channel. The attack phase continues until the amplitude reaches the level given by the parameter ALA.

For reference the six parameters are defined again here.

Parameter	Range	Function
A	-127 to 127	Change of amplitude per step during attack phase
AD	-127 to 127	Change of amplitude per step during decay phase
AS	-127 to 0	Change of amplitude per step during sustain phase
AR	-127 to 0	Change of amplitude per step during release phase
ALA	0 to 126	Target level at end of attack phase
ALD	0 to 126	Target level at end of decay phase

In our example the attack phase takes four seconds and each step lasts 1/4 seconds so there will be 16 steps. We want these 16 steps to get us from an amplitude of zero to an amplitude of at least 100 – if we make each step increase the amplitude by seven we will get there in 16 steps. So parameter AA = 7.

During the decay phase the amplitude must drop from 100 to 60 in two seconds. During two seconds there are eight steps. So the amplitude drops 40 units (100-60) in eight steps – so each step must reduce the amplitude by five units. Thus AD=-5. So far we have determined the following parameters of the amplitude envelope.

A=7

AD=-5

ALA=100

ALD=50

In our case the amplitude does not change during the sustain period so we can set AS = 0. The sound will go on until the sustain phase is ended. The total time allowed for the attack, decay and sustain phases is given by the duration part of the SOUND command. The release phase then starts.

Note that the length of the attack and decay phases is set by the values chosen for AA, AD, ALA and ALD but that the sustain phase can be terminated either by the amplitude reaching zero or the time set by the duration of the SOUND statement running out. The duration has to be set with care to ensure that it doesn't cut the note off at the wrong moment.

At the end of the sustain period the note enters the release phase where the note changes in amplitude at the rate set by AR until it reaches zero.

As you may have guessed there are many ways for things to go wrong so that a phase does not complete as expected. For example with ALA set to 100 and ALD set to 50 and a decay rate (AD) of zero the amplitude will not decay at all during the decay phase. However the sound will be moved to the release phase when the duration is reached.

The **ENVELOPE** statement is very complicated and there is a wide range of possible effects. You will have to use it a lot before you can accurately predict what effect you will produce.

Some sample envelopes to try out:

```
ENVELOPE 1,1,0,0,0,0,0,0,2,0,-10,-5,120,0
ENVELOPE 2,3,0,0,0,0,0,0,121,-10,-5,-2,120,120
ENVELOPE 3,7,2,1,1,1,1,1,121,-10,-5,-2,120,120
ENVELOPE 4,1,0,0,0,0,0,0,61,0,-10,-120,120,0
ENVELOPE 1,8,1,-1,1,1,1,1,121,-10,-5,-2,120,1
```

Note synchronisation and other effects

The first parameter of the **SOUND** statement has been considered, up to now, to control only the channel number. It can in fact control a number of other features. For this purpose the channel number should be considered as a four digit hexadecimal number

C=&HSFN

Parameter	Range	Function
N	0 to 3	Channel number itself
F	0 or 1	Flush control
S	0 to 3	Synchronisation control
H	0 or 1	Continuation control

N selects the channel number.

F If F is 0 the sound will be placed in a channel queue if a note is playing on that channel. If F = 1 then the channel queue will be flushed (emptied) so that the sound can be generated immediately.

S It is possible to synchronise two or more channels so that they do not start until all have received a note marked for synchronous production. The value of S determines how many other channels are to form the chord. Thus for a three note chord all three channels should be fed a note with S set to 2.

H This parameter allows the previous effect on the channel to continue if it is set to 1. In this case the amplitude, pitch and duration parameters of the new sound command have no effect. Because the 'dummy' note thus created is added to the queue in the normal way it can be used to ensure that the release

phase of a sound is completed. Normally the release phase is truncated by the next note on the queue. If H=0 then the note is treated as a 'real' note in the usual way.

Typical values of C are:

C=&201 A note on channel 1 to be synchronised with two others.
C=&12 A note on channel 2 is to be played immediately regardless of what was in the channel 2 queue.

A more succinct description of `SOUND` and `ENVELOPE` is given in the BASIC keywords chapter.

31 File handling

You are probably already aware that as well as storing computer programs on cassette or disc, you can store 'data'. By 'data' we mean sets of numbers or words. We might, for example, store a set of names and telephone numbers. This set of data is called a file. A set of BASIC statements are provided to enable you to read information from files, to write information to a new file, to update an existing file, to delete a file, to find out how big a file is, to move to a certain record in a file, to check if you have reached the end of a file and to obtain a catalogue of all the files that exist. There are also several other statements for performing other actions on the files.

One of the major features of the BBC Microcomputer system is that exactly the same statements are used no matter which 'filing system' is in use. A number of different filing systems are available including cassette, disc and Econet systems. Programs written to work on a cassette filing system will usually work unmodified on a disc system. See chapter 36 for details of other filing systems.

First here is a summary of all the file handling statements available in BBC BASIC.

*CAT	Gives a catalogue of all data files and programs on the cassette or floppy disc. It takes a very long time on a cassette.
OPENIN	Opens a file so that it can be read.
OPENOUT	Opens a new (empty) file for writing.
OPENUP	Opens a file for reading and/or writing. (Not cassette.)
INPUT#	Reads data from a file into the computer.
PRINT#	Writes data from the computer into a file.
BGET#	Reads a single character (byte) from a file.
BPUT#	Writes a single character (byte) to a file.
PTR#	Can be used either to find out which record we are about to read (or write) or to move to a specific record. (Not cassette.)
EXT#	Indicates how long a file is. (Not cassette.)
EOF#	Indicates whether or not the end of a file has been reached.
CLOSE#	Indicates to the computer that you have finished with a file.

The statement ***CAT** can be used at any time. However before you can use any of the other file statements you have to *open* the file. After you have opened a file you can read and write to it as much as you wish. When you have finished with the file you must close it.

An analogy may help to make one or two points clearer. The files are all kept in one room and your only method of communicating with them is via five telephone links to five clerks. In addition there is a supervisor who knows which telephone line to use to communicate with the right clerk. It is the clerk's job to keep all the files organised and you really have no idea how he or she looks after the files – nor does it matter, so long as the method is efficient.

To return to our computer, suppose that we wish to create a file called **"DRINKS"** in which we list every drink we have ever tried. First of all we have to ask the supervisor to allocate a phone line and clerk to us. The statement

```
X = OPENOUT "DRINKS"
```

will place the number of the channel (telephone line) allocated to the file into the variable **X**. Next we can ask the user for the names of the drinks using

```
INPUT "WHAT IS THE DRINK CALLED?", D$
```

and then send the name (held in **D$**) to the clerk to be entered in the file. Notice how we have to use the variable **X** to ensure that it is entered in the correct file.

```
PRINT#X,D$
```

We could then repeat this process until the user replied with the word **STOP**. The program would look like this:

```
10 X=OPENOUT "DRINKS"
20 REPEAT
30 INPUT "What is the drink called ", D$
40 PRINT#X, D$
50 UNTIL D$="STOP"
60 CLOSE# X
```

When run the program will save the data on cassette (if one is connected).

```
>RUN
What is the drink called?WHISKY
What is the drink called?VODKA
What is the drink called?GIN
What is the drink called?WINE
What is the drink called?CIDER
What is the drink called?TOMATO JUICE
What is the drink called?STOP
```

That has created a file called **"DRINKS"** which has been stored on cassette. The program to read the file back in is also straightforward.

```
10 Y=OPENIN "DRINKS"
20 REPEAT
30 INPUT Y , R$
40 PRINT R$
50 UNTIL R$="STOP"
60 CLOSE# Y
```

When this is run, the list will appear on the screen as it is read from the cassette.

```
>RUN
WHISKY
VODKA
GIN
WINE
CIDER
TOMATO JUICE
STOP
```

For most applications that is all you will need to know about file handling and you will only use statements like these

```
*CAT
X=OPENIN "FILENAME"
X=OPENOUT "FILENAME"
PRINT#X,A,B$
INPUT#X,A,B$
CLOSE#X
```

"FILENAME" is the name of the file which normally consists of up to ten letters but see chapter 35 for more details.

A and **B$** represent any (and as many) numeric and string variables as you wish to record.

X is a numeric variable used to remember the channel number allocated to the file number.

For more specialised applications a number of other functions and statements are provided. **BGET#** and **BPUT#** enable single characters to be input and output. They would be used for recording special data, for example, laboratory experiments.

EXT# and **PTR#** are used with disc systems where random access files are required. They cannot be used with cassette systems.

EOF# enables a program to detect the end of the file when reading in data. It is normally used in the following way

```
10 Y=OPENIN "DRINKS"
20 REPEAT
30 INPUT#Y, A$
40 PRINT A$
50 UNTIL EOF#Y
60 CLOSE#Y
```

Telephone book program

One of the programs on the WELCOME cassette can be used to keep a personal telephone directory. Clearly it should be possible to save a copy of all your entries on to cassette and to load them back into the computer later. Several modifications must be made to the program to enable this to happen. These modifications are shown below. Once you have modified the program you can then save the corrected version with a new name, for example

```
SAVE "TELE2"
```

First, load the program. Don't **RUN** it, but type **CTRL N** then **LIST** to list the program in 'page mode'. To move down the program, press **SHIFT**. When you come to a page requiring one of the changes set out below, press **ESCAPE** and edit the line in the normal way.

Lines 210 to 280 omit final **'**
Add new lines:

```
282 PRINT" 9 - Load data from cassette"
284 PRINT" 0 - Save data to cassette"
```

Change line 290 to

```
290 SEL = -1
```

In line 300 change **TAB(3,22)** to **TAB(3,23)**
Line 330 change to

```
330 IF A<0 OR A>9 THEN 310
```

Change line 350 to

```
350 IF SEL> -1 THEN PRINT TAB(2,SEL+3 -
10*(SEL=0));CHR$(&89);
```

Change line 360 to

```
360 PRINT TAB(2,A+3-10*(A=0));CHR$(&88);A;CHR$(&89)
```

Change line 380 to

```
380 IF SEL=-1 THEN 300
```

Change line 500 to

```
500 ON SEL + 1 GOTO
505,510,520,530,540,550,560,570,580,590
```

New lines:

```
505 PROCSAVE:GOTO 200
590 PROCLOAD:GOTO 200


10000 DEF PROCLOAD
10005 PRINT TAB(0,16); "Play tape and press any key"
10007 Q=GET
10008 PRINT "Please wait"
10010 E%=OPENIN "DATA"
10020 INPUT#E%,X
10030 FOR I%=1 TO X
10032 INPUT#E%,NAME$(I%),PHONE$(I%)
10034 PROCPACK (I%)
10036 NEXT
10040 CLOSE#E%
10050 ENDPROC
11000 DEF PROCSAVE
11005 PRINT TAB(0,16); "Please press ";
11010 E%=OPENOUT "DATA"
11015 PRINT '"Please wait"
11020 PRINT#E%,X
11030 FOR I%=1 TO X
11032 PRINT#E%,NAME$(I%),PHONE$(I%)
11036 NEXT
11040 CLOSE#E%
11050 ENDPROC
```

32 Speeding up programs and saving memory space

For some applications it is important that a program runs as quickly as possible and a few tips are given here which will, together, substantially increase the execution speed of programs. In other applications space may be at a premium and other suggestions are given for saving space. Sometimes there is a trade-off between the size of a program and speed and the user will have to decide which is more important.

The most dramatic saving that can be made is in the speed of execution of programs. The use of integer variables (eg `WEIGHT%`), and especially of the resident integer variables `A%` to `Z%`, will result in execution times as little as 50% of those achieved with 'real' variables. Again, integer division (`DIV`) is much faster than normal division when working with integers. Using integer arrays rather than real arrays will save 20% of the memory required.

Execution speed can also be increased in the following ways.

1. Allocate variable names with an even spread throughout the alphabet – so don't start all your variables with 'F', for example.

2. Omit the control variable after the word `NEXT` – eg say `NEXT` rather than `NEXT X`. This saves a reasonable amount of time.

3. `REPEAT...UNTIL` loops are much faster than `IF...THEN...GOTO` loops.

4. Procedures are faster than `GOSUB`s, and it is faster to pass parameters to a procedure than to use global variables – ie do use `PROCBOX(X,Y,Z)` rather than `PROCBOX`.

5. If you have a line which contains a lot of 'integer' arithmetic and a little 'real' arithmetic then, if possible, place the integer work at the start of the line where it will be executed first.

6. Have as few line numbers as possible – ie use long lines and spread the line numbers out rather than re-numbering with an interval of 1. An interval of 10 is good.

As far as space saving is concerned the following can be tried – but both reduce the readability of programs and should not be used unless it is really necessary.

7. Omit spaces wherever possible – but you must keep a space or a **%** or **$** sign or some other separator before most keywords to avoid ambiguity. If a variable **FRED** is in use then you must write

Y=FRED OR MARY

and not

Y=FREDORMARY

In the latter case the computer will look for the variable **FREDORMARY** rather than the two variables **FRED** and **MARY**. The space after **OR** is not required.

8. Omit **REM** statements.

9. Remember that the whole of user memory can be kept for use by your programs by using the 'shadow screen' facility – see chapter 42 for more details.

33 BASIC keywords

This chapter contains a detailed description of every word that BASIC understands. These words are called 'keywords'.

Some parts of the description are intended for the novice user and others for the person who is familiar with BASIC. Note that your BBC Microcomputer is supplied with the latest version of BBC BASIC, known as BASIC II. If you are already familiar with the earlier version, you may wish to refer to chapter 49 before reading this chapter. Chapter 49 lists those BASIC keywords which exist only in BASIC II. Each keyword is described under a number of headings as follows:

Keyword

Sometimes followed by a few words explaining the derivation of the word.

Purpose

A plain-English description of what the keyword does. This is intended for the person who is learning BASIC.

The only technical terms used are 'string' and 'numeric' – if you don't understand those two words then read chapter 9 first. The mathematical functions `SIN`, `COS`, `TAN` etc are not exlained for the absolute beginner – there just isn't room to explain everything!

Examples

This section gives a few one-line examples of the keyword (not complete programs). Some of the examples have a number at the start of the line. This number is an 'example line number'.

The examples are only intended to be illustrative. In some cases a line of BASIC program may overflow onto the next line as elsewhere in this book.

Description

In this section the keyword is described using normal computer jargon.

Syntax

The syntax of each keyword's usage is given more by way of helpful explanation than for its strict accuracy. Purists will, rightly, complain at travesty of Backus-Naur form. Others may find the entries useful.

The following symbols are used as part of the syntax explanation:

{ }	Denote possible repetition of the enclosed symbols zero or more times.
[ ]	Enclose optional items.
|	Indicates alternatives from which one should be chosen.
<num-const>	Means a numeric constant such as '4.5' or '127'
<num-var>	Means a numeric variable such as 'X' or 'length'
<numeric>	Means either a <num-const> or a <num-var>, or a combination of these in an expression such as `"4*X+6"`
<string-const>	Means a string enclosed in quotation marks, eg `"ROBERT BULL"`.
<string-var>	Means a string variable such as `A$` or `NAME$`.
<string>	Means either a <string-const> or a <string-var>, or an expression such as `A$+"LINDA"`.
<testable condition>	Means something which is either `TRUE` or `FALSE`. Since `TRUE` and `FALSE` have values it is possible to use a <numeric> at any point where a <testable condition> is required. The distinction between these two is, in fact, rather unnecessary.
<statement>	Means any BASIC statement, for example, `PRINT` or `GOSUB` or `PROC`.
<variable name>	Means any sequence of letters or numbers that obeys the rules for variables (see chapters 3, 9 and 21).

Associated keywords

This section is intended to draw your attention to other keywords which either have similar functions or which are normally used in conjunction with this keyword. You will probably find it helpful to read the pages which describe the associated keywords.

Demonstration program

If appropriate a short program is included to illustrate the use of the keyword. Parentheses are generally optional where sense is unaffected.

ABS absolute value

Purpose

This function turns negative numbers into equivalent positive numbers but leaves positive numbers alone. For example the absolute value of -9.75 is 9.75 while the absolute value of 4.56 is 4.56.

The `ABS` function is often used when calculating the difference between two values if you do not know which is the larger of the two. Thus (K-L) will be positive if K is greater than L, and will be negative if L is greater than K.

For example, if K = 9 and L = 12 then (K - L) would be equal to -3. However the value of `ABS(K-L)` will always be positive. In the example given `ABS(K-L)` would equal 3.

Examples

```
205 error=ABS(DIFFERENCE)

100 DIFF=ABS(X2-X1)

PRINT ABS(temp%-50)
```

Description

A function giving the absolute value of its argument.

Syntax

<num-var>=**ABS**(<numeric>)

Associated keywords

`SGN`

ACS arc-cosine

Purpose

To calculate an angle whose cosine is known. The calculated angle is in radians but may be converted to degrees by using the function **DEG**. See **DEG** for more information.

Examples

```
10 X=ACS(Y)

1205 angle=DEG(ACS(0.5678))

330 OUT=ACS(.234)

PRINT ACS (0.5)
```

Description

A function giving the arc-cosine of its argument. The result is in radian measure.

Syntax

<num-var>=**ACS**(<numeric>)

Associated keywords

ASN, ATN, SIN, COS, TAN, RAD, DEG

ADVAL analogue to digital converter value

Purpose

An analogue signal is one which can have almost any value – including fractional parts. It is contrasted with a digital signal which is expressed in exact numbers. The height of the water in a harbour is an analogue quantity whereas the number of boats it contains is a digital quantity.

Watches always used to have analogue dials – 'The time is about four fifteen'. Electronic things usually work with whole numbers; for example

16h : 15m : 23s

There are four 'analogue to digital' converters in the BBC Microcomputer. Each analogue to digital converter in the computer accepts a voltage and gives out a whole number indicating how large the voltage is. This voltage might be controlled by, for example, the position of a 'games paddle' or 'joystick' control which is connected to the computer. Alternatively the computer might be connected to a speed sensor on a piece of machinery or it might measure the temperature of a room.

The input voltage range is 0 volt to 1.8 volt. When the input is 0 V the converter produces the number zero. With 1.8 V input the converter produces the number 65520. Why 65520? The circuit in the computer which does the conversion was designed to give out numbers in the range 0 to 4095. However it may well be that future converters can give out numbers over a larger range – enabling the computer to measure things more accurately. In order to ensure that the BBC Microcomputer can be used in this situation we have specified a large range.

Instead of producing numbers in the 0 to 4095 range it produces a number in the range 0 to 65520. Therefore instead of numeric results going up in the sequence 0, 1, 2, 3 etc they will go 0, 16, 32, 48, 64 etc. If you prefer the range 0 to 4095 then just divide the value by 16.

There are four analogue input channels provided in the BBC Microcomputer and the number in parentheses after the keyword **ADVAL** refers to the channel whose value you wish to find. The channels are numbered 1, 2, 3, 4.

ADVAL(0) performs a special function in that it can be used to test to see which of the 'fire' buttons is pressed on the games paddles. The value returned also indicates which ADC channel was the last one to be updated. The following can be used to extract these two pieces of information from the value returned by **ADVAL(0)**.

```
X=ADVAL(0) AND 3
```

will give a number with the following meaning

X=0 no button pressed
X=1 left side fire button pressed
X=2 right side fire button pressed
X=3 both fire buttons pressed

```
X=ADVAL(0) DIV256
```

will give the number of the last analogue to digital channel to complete conversion. If the value returned is zero then no channel has yet completed conversion.

ADVAL with a negative number in the parentheses, eg **X=ADVAL(-3)**, can be used to see how full any of the internal buffers are. When characters are typed in on the keyboard they are put into a buffer from which they are extracted with statements like **INPUT** and **GET**. Other buffers are used internally for other purposes. The exact meaning of the number returned depends on the buffer being tested.

X=ADVAL(-1)   Returns the number of characters in the keyboard buffer.

X=ADVAL(-2)   Returns the number of characters in the RS423 input buffer.

X=ADVAL(-3)   Returns the number of free spaces in the RS423 output buffer.

X=ADVAL(-4)   Returns the number of free spaces in the printer output buffer.

X=ADVAL(-5)   Returns the number of free spaces in the sound channel 0 buffer.

X=ADVAL(-6)   Returns the number of free spaces in the sound channel 1 buffer.

X=ADVAL(-7)   Returns the number of free spaces in the sound channel 2 buffer.

X=ADVAL(-8)   Returns the number of free spaces in the sound channel 3 buffer.

X=ADVAL(-9)   Returns the number of free spaces in the speech buffer.

This feature can be used, for example, to ensure that a program never gets stuck waiting for a sound channel to empty, eg:

```
IF ADVAL(-7)<> 0 THEN SOUND 2,...etc
```

Examples

```
980 X=ADVAL(3)

125 TEMP=ADVAL(X)

intensity=ADVAL(1)
```

Syntax

<num-var>=**ADVAL**(<numeric>)

Description

A function which returns the last known value of the analogue to digital channel given in its argument. There are four channels, each of 10 bit resolution, but the returned value is scaled to 16 bits.

The analogue to digital converter cycles repeatedly through the selected channels and keeps a table of the result so that the function **ADVAL** returns very quickly. New samples are taken about every ten milliseconds. Therefore with four channels selected results will be updated every 40ms. See chapter 43 for information on changing the number of channels selected.

AND

Purpose

AND can be used either as a logical operator or as a 'bit by bit', or 'Boolean' operator.

As a logical operator **AND** is used to ensure that two conditions are met before something is done. For example

```
IF X=9 AND Y=0 THEN PRINT "HELLO"
```

Logical **AND** is most often used as part of an **IF...AND...THEN...** construction.

Boolean **AND** compares the first bit of one number with the first bit of another number. If both bits are set to a one (rather than a zero) then the first bit in the answer is also set to a one. This process is then repeated for the second bit in each of the two numbers being compared and so on for all 32 bits in the numbers. For example the result of **14 AND 7** is 6, since in binary

```
    14 is 0000 0000 0000 0000 0000 0000 0000 1110
     7 is 0000 0000 0000 0000 0000 0000 0000 0111
     6 is 0000 0000 0000 0000 0000 0000 0000 0110
```

Examples

```
300 IF length>9 AND wt>9 THEN PRINT "YES"
```

```
100 IF X=2 AND cost>5 AND J=12 THEN PRINT "NO!!"
```

The above example will only print **NO!!** if all three conditions are met.

Description

The operation of Boolean **AND** between two items. Note that the logical and Boolean operations are in fact equivalent. This follows since the value of **TRUE** is -1 which is represented on this machine by the binary number

1111 1111 1111 1111 1111 1111 1111 1111

Similarly the binary value of **FALSE** is

0000 0000 0000 0000 0000 0000 0000 0000

Thus **PRINT 6=6** would print **-1** since **6=6** is **TRUE**.

Syntax

<num-var> = <numeric> **AND** <numeric>

or

<testable condition> = <testable condition> **AND** <testable condition>

Associated keywords

EOR, **OR**, **FALSE**, **TRUE**, **NOT**

ASC American Standard Code (ASCII)

Purpose

There are two commonly used methods of talking about characters (things like A, B, 5, ?, and so on). Obviously they are single characters! So we can say `D$="H"` - meaning put the letter H into the box in the computer labelled D$. The computer understands this but it doesn't actually put an H into the box, Instead it stores a number which represents the letter H (in fact the number is 72). Every character has a unique corresponding number called its ASCII code. (ASCII stands for American Standard Code for Information Interchange. The abbreviation ASCII rhymes with 'Laski'.)

Sometimes it is convenient to find out what number corresponds to a particular character - that is its ASCII code. You can look it up at the back of this book or you can say to the computer

`PRINT ASC("H")`

The function `ASC` gives the ASCII value of the first letter in the string. Thus

`PRINT ASC("Good")`

gives 71, the ASCII value of 'G'.

The reverse process of generating a one-character string from a given ASCII value is performed by the function `CHR$`.

Examples

`25 X=ASC("Today")`

would put the ASCII value of 'T' which is 84 into the variable X.

`650 value5=ASC(A$)`

Description

A function returning the ASCII character value of the first character of the argument string. If the string is null (empty) then -1 will be returned.

Syntax

<num-var> = `ASC`(<string>)

Associated keywords

`CHR$`, `STR$`, `VAL`

ASN arc-sine

Purpose

To calculate an angle whose sine is known. The calculated angle is in radians but may be converted to degrees by using the function **DEG**. A radian is equal to about 57 degrees. Mathematicians often prefer to work in radians.

Examples

```
340 J=ASN(0.3456)

30 angle=DEG(ASN(.7654))

PRINT ASN(.5)
```

Description

A function giving the arc-sine of its argument. The result is in radian measure.

Syntax

<num-var>=**ASN**(<numeric>)

Associated keywords

ACS, **ATN**, **SIN**, **COS**, **TAN**, **RAD**, **DEG**

ATN arc-tangent

Purpose

To calculate an angle whose tangent is known. The calculated angle is in radians but may be converted to degrees by using the function `DEG`.

Examples

```
1250 X=ATN(Y)
```

```
240 value=DEG(ATN(22.31))
```

Description

A function giving the arc-tangent of its argument. The result is in radian measure.

Syntax

<num-var>=**ATN** (<numeric>)

Associated keywords

ACS, **ASN**, **SIN**, **COS**, **TAN**, **RAD**, **DEG**

AUTO automatic

Purpose

When typing a BASIC program into the computer it is common to make the first line of the program line number 10, the second line 20 etc. To save having to type in the line number each time, the command **AUTO** can be used to make the computer 'offer' each line number automatically. Used on its own the command **AUTO** will offer first line 10 and then lines 20, 30, 40 etc. The command **AUTO 455** would instead start the process at line number 455, followed by lines 465, 475, 485 etc.

Another option allows the user to select the step size. Thus the command **AUTO 465,2** would cause the computer to offer lines 465, 467, 469, 471 etc. The larges step size is 255.

To escape from **AUTO** mode the user must press the key marked **ESCAPE**. **AUTO** mode will be abandoned if the computer tries to generate a line number greater than 32767.

Examples

```
AUTO

AUTO 220

AUTO 105,5
```

Syntax

AUTO [<num-const>[**,**<num-const>]]

Description

AUTO is a command allowing the user to enter lines without first typing in the number of the line. Because **AUTO** is a command it cannot form part of a multiple statement line. **AUTO** mode can be left by pressing **ESCAPE** or generating a line number exceeding 32767.

AUTO may have up to two arguments. The first optional argument gives the starting line number and the second optional argument gives the increment between line numbers.

BGET# get a byte from file

Purpose

Numbers and words can be recorded on cassette tape and on disc. The function **BGET#** enables a single character or number to be read back into the computer from the cassette, disc or network. Before using this statement a file must have been opened using the **OPENIN** function or else an error will occur (see chapter 31 for more information about 'files'). When a file is opened, using **OPENIN**, the computer will allocate the file a channel number. This number must be used in all subsequent operations on the file, for example when reading the file or when writing a new file. Again see the chapter on file handling for more information.

Examples

```
6000 character=BGET# (channel)
```

```
340 next_letter%=BGET#C
```

Description

A function which gets a byte from the file whose channel number is the argument. The file must have been opened before this statement is executed.

Syntax

<num-var>=**BGET#**<num-var>

Associated keywords

OPENIN, **OPENUP**, **OPENOUT**, **CLOSE#**, **EXT#**, **PTR#**, **PRINT#**, **INPUT#**, **BPUT#**, **EOF#**

BPUT# put a byte to file

Purpose

To store a byte on cassette or disc. See chapter 31 for a more detailed description of file handling. The number which is sent to the file can have any value between 0 and 255. If you attempt to store a number that is greater than 255, then 256 will be repeatedly subtracted from the number until it is less than 256. The final number will then be sent to file. (This statement is used to store single bytes – not large numbers. Larger numbers can be stored using **PRINT#**.) As with **BGET#** the file must be 'open' before this statement can be used.

Examples

```
30 BPUT# channel,number
```

```
700 BPUT#N,32
```

```
450 BPUT# STAFF_FILE, A/256
```

Description

A statement which puts a byte to the file whose channel number is the first argument. The second argument's least significant byte is sent. The file must be open before this statement is executed.

Syntax

BPUT#<num-var>**,**<numeric>

Associated keywords

OPENIN, **OPENUP**, **OPENOUT**, **CLOSE#**, **EXT#**, **PTR#**, **PRINT#**, **INPUT#**, **BGET#**, **EOF#**

CALL transfer control to a machine code subroutine

Purpose

This statement makes the computer execute a piece of machine code which the user has previously placed in the computer's memory. Before using this powerful statement you should have a good understanding of machine code and assembly language as incorrect use can destroy a program completely! Unfortunately there is not enough room in this book to teach assembly language programming but brief guidance on the principles of 6502 assembly language is given in chapter 44.

Examples

```
50 rotate=&0270
60 CALL rotate,J,K,L

200 CALL 1234,A$,M,J$
```

Description

A statement to call a piece of machine code. The number of parameters passed is variable and may be zero. The parameters are variable parameters and may be changed on execution of the subroutine. The addresses of parameters are passed in a *parameter block* starting at location 0600 hex.

On entry to the subroutine the processor's A, X, Y registers are initialised to the least significant bytes of the integer variables A%, X% and Y%. The carry flag is set to the least significant bit of the C% variable.

On entry a parameter block is set up by the computer and it contains the following list:

Number of parameters	1 byte
Parameter address	2 bytes
Parameter type	1 byte
Parameter address	} Repeated as often
Parameter type	as necessary

Parameter types

0 - 8 bit byte (eg ?X)
4 - 32 bit integer variable (eg !X or X%)
5 - 40 bit floating point number (eg V)
128 - A string at a defined address (eg $X - terminated by a &0D)
129 - A string variable such as A$

In the case of a string variable the parameter address is the address of a *string information block* which gives the start address, number of bytes allocated and current length of the string in that order.

Syntax

`CALL`<numeric>`{,`<num-var>`|`<string-var>`}`

Associated keywords

`USR`

CHAIN

Purpose

CHAIN enables a program to be split up into a number of small sections.

The **CHAIN** statement is used to enable one program to **LOAD** and **RUN** another program automatically. For example, one program might enable the user to enter the number of hours worked by employees and that program might **CHAIN** a second program which would print out the payslip. In turn that might **CHAIN** a third program which would do a cost per hour analysis on the data held on the file.

CHAIN is also useful in a game with a lot of instructions. The instructions could all be stored as one file which would then **CHAIN** the main game – thus releasing a lot of the computer's memory.

CHAIN"" (without the program name) will chain the next program on a cassette, whatever its file name. This will not work with disc filing systems where you must give the file name. For that reason it must not be used in programs which may be used on disc systems.

Examples

```
900 CHAIN "GAME_1"
```

```
1234 CHAIN "NEWPROG"
```

```
CHAIN A$
```

Description

A statement which will load and run the program whose name is specified in the argument. All variables except @% and A% to Z% are cleared.

Syntax

CHAIN<string>

Associated keywords

LOAD, SAVE

CHR$ character string

Purpose

To generate a character (single letter or number etc) from the number given in string form. The character generated will be the ASCII character at the position given in the ASCII table. See the description of **ASC** and the full ASCII table in Appendix C.

The statement **VDU** has a similar effect to **PRINT CHR$** and may be more useful in some applications.

Examples

```
220 RED$=CHR$(129)
```

```
1070 PRINT CHR$(8);
```

makes the cursor move left one position.

```
PRINT CHR$ (7)
```

causes a short note to be emitted by the loudspeaker.

Description

A string function whose value is a single character string containing the ASCII character specified by the significant byte of the numeric argument. Thus **CHR$(-1)** would give ASCII character number 255.

Note that the statement **VDU** is probably more useful when sending characters to the screen, since it involves less typing. **CHR$** is needed when you wish to put a special character into a string.

Syntax

<string-var>=**CHR$**(<numeric>)

Associated keywords

ASC, **STR$**, **VAL**, **VDU**

CLEAR

Purpose

This tells the computer to forget all variables previously in use, including string variables and arrays but excluding the 'resident integer variables' **@%** and A% to Z% which are not affected in any way. See chapter 9 for an explanation of integer and string variables.

Examples

```
350 CLEAR
```

```
CLEAR
```

Description

A statement which deletes all variables except the resident integer numeric variables **@%** and A% to Z%

Syntax

```
CLEAR
```

Associated keywords

None

CLG clear the graphics screen

Purpose

To clear the graphics area of the screen. The graphics area of the screen is left in the colour selected as the 'current graphics background colour'. See the keyword `GCOL` for more information. The graphics cursor is then moved to its home position (0,0) which is at the bottom left of the graphics area.

Examples

```
870 CLG
```

```
CLG
```

Description

Clears the current graphics area of the screen and sets this area to the current graphics background colour in addition. The statement then moves the graphics cursor to the graphics origin (0,0).

Syntax

```
CLG
```

Associated keywords

`CLS, GCOL`

CLOSE#

Purpose

To inform the computer that you have completely finished with a particular file. The computer then transfers any data still in memory to cassette, disc or Econet as needed. See chapter 31 on file handling for more information.

Example

`90 CLOSE#N`

Description

A statement used to close a specific disc or cassette file. `CLOSE# 0` will close all files.

Syntax

`CLOSE#`<numeric>

Associated keywords

OPENIN, OPENUP, OPENOUT, EXT#, PTR#, PRINT#, INPUT#, BGET#, BPUT#, EOF#

CLS clear the text screen

Purpose

To clear the text area of the screen. Any graphics in this area will also be cleared. The text area will be left in the 'current text background colour'. The text cursor will then be moved to its 'home' position at the top left of the text area. See the keyword **COLOUR** for more information about text background colours.

Examples

```
560 CLS
```

```
CLS
```

Description

Clears the current text area and sets this area of the screen to the current text background colour. The statement then causes the text cursor to move to the text origin (0,0) at the top of the current text area. **CLS** resets **COUNT**.

Syntax

```
CLS
```

Associated keywords

CLG, COLOUR

COLOUR

Purpose

This statement selects the colour in which the computer is to print the text and also its background. The command has a number of variations which are most easily explained by example.

Type in the following:

```
MODE 5
COLOUR 1
COLOUR 2
COLOUR 3
```

and press **RETURN** at the end of each line as usual.

As you will have seen, these commands change the colour of the text. This is often called the 'text foreground colour'. Now try

```
COLOUR 129
COLOUR 130
COLOUR 128
```

These numbers change the 'text background colour'.

In any two colour **MODE (MODE 0, 3, 4** or **6)** the following normally apply:

Foreground	Background	Colour
0	128	Black
1	129	White

In any four colour **MODE (MODE 1** or **5)** the following normally apply:

Foreground	Background	Colour
0	128	Black
1	129	Red
2	130	Yellow
3	131	White

In **MODE 2** the following normally apply:

Foreground	Background	Colour
0	128	Black (normal background)
1	129	Red
2	130	Green
3	131	Yellow
4	132	Blue
5	133	Magenta (blue/red)
6	134	Cyan (blue/green)
7	135	White (normal foreground)
8	136	Flashing black/white
9	137	Flashing red/cyan
10	138	Flashing green/magenta
11	139	Flashing yellow/blue
12	140	Flashing blue/yellow
13	141	Flashing magenta/green
14	142	Flashing cyan/red
15	143	Flashing white/black

If you are not familiar with BASIC then you may already have had too much of this! Nevertheless, it is possible, for example in a four colour **MODE** to select any four colours from the available 16 effects by using another command. Remember that the colours given above (black, red, yellow, white) will be available as soon as four colour **MODE** is selected – but you can then select other colours later.

Try the following:

```
MODE 5
COLOUR 1
VDU 19,1,4,0,0,0
VDU 19,1,5,0,0,0
COLOUR 2
VDU 19,2,4,0,0,0
VDU 19,1,3,0,0,0
```

As you will see the statement **VDU 19**, can be used to change the 'actual colour' of colour 1 or 2.

The number which follows the **VDU 19**, is the number that is referred to by the **COLOUR** statement. It is referred to as a 'logical colour'.

The number which follows the 'logical colour' is referred to as the 'actual colour' and is as follows:

0	Black
1	Red
2	Green
3	Yellow
4	Blue
5	Magenta (blue/red)
6	Cyan (blue/green)
7	White
8	Flashing black/white
9	Flashing red/cyan
10	Flashing green/magenta
11	Flashing yellow/blue
12	Flashing blue/yellow
13	Flashing magenta/green
14	Flashing cyan/red
15	Flashing white/black

Thus the statement `VDU 19,3,6,0,0,0` will set logical colour 3 to be cyan. So if in `MODE 4`, a two colour `MODE`, you wanted black letters on a yellow background you would issue the command:

```
VDU 19,1,0,0,0,0
VDU 19,0,3,0,0,0
```

Alternatively, you could string the whole lot together as

```
VDU 19,1,0,0,0,0,19,0,3,0,0,0
```

This combination of the `COLOUR` statement and the `VDU 19` statement enables a very wide range of effects to be obtained. There are also calls which enable the flash rates of the colours to be altered as well. See chapter 43 on `*FX` calls.

Syntax
`COLOUR`<numeric>

Associated keywords
`VDU`, `GCOL`

COS cosine

Purpose

To calculate the cosine of an angle. Note that the number in parenthesis (the angle) is expressed in radians and not in degrees. To convert from degrees to radians use the function **RAD**.

Examples

```
PRINT COS(2.45)

780 X=COS(Y)

655 Number=COS (RAD (45))
```

Description

A function giving the cosine of its argument. The argument must be given in radians.

Syntax

<num-var>=COS(<numeric>)

Associated keywords

SIN, TAN, ACS, ASN, ATN, DEG, RAD

COUNT

Purpose

COUNT counts all the characters printed using **PRINT**, whether to screen, printer or RS423 output channel. On the other hand **POS** returns the current position of the actual text cursor on the screen.

Examples

```
290 A=COUNT
```

```
75 fred=COUNT
```

```
PRINT COUNT
```

Description

A function returning the number of characters printed since the last new line. **COUNT** is set to zero if the output stream is changed.

Syntax

<num-var>=COUNT

Associated keywords

POS

Demonstration program

```
 5 REM to print a row of 16 * signs
 7 REM this is not the easiest way!
10 X=16
20 REPEAT PRINT "*";
30 UNTIL COUNT=X
```

DATA

Purpose

DATA is used in conjunction with the keyword **READ**, and sometimes with **RESTORE**, to enable you to make available automatically any data (numbers and words) that will be needed by a program.

For example, if you were writing a geography quiz, you might want to use the names of five countries and their five capital cities each time you used the program. The names of the cities and countries can be entered as **DATA** in the program and will always be there when the program is run.

Computers using the language BASIC are really clumsy at handling information like this, as the demonstration program on the next page shows.

In the example program the **DATA** consists of many words. **DATA** statements can just as well contain numbers – or a mixture of words and numbers. In our example the words were all read into a string array.

It is essential that the **DATA** contains numbers where numeric variables are to be filled. Text information, eg **hello**, will just give 0.

There is no need to put each word in quotes unless leading spaces are important in the **DATA** words, for example **"   four spaces"**.

If you wish to have leading spaces then these words should be enclosed in quotes. Since a comma is used to separate items of **DATA**, if you want a comma in your **DATA**, you must enclose the **DATA** in quotes.

Examples

```
100 DATA "Allen, Stephen", Stamp dealer, 01-246
8007, 24

130 DATA "TOP OF ROOF", 450, January
```

Description

A program object which must precede all lists of data intended for use by the **READ** statement.

Syntax

DATA<str-const>|<num-const>{,str-const>|<num-const>}

Associated keywords

READ, RESTORE

Demonstration program

```
 10 REM geography quiz
 20 DIM  city$(5)
 30 DIM country$(5)
 40 FOR x=1 TO 5
 50 READ city$(x)
 60 READ country$(x)
 70 NEXT x
 80 right=0
110 FOR x=1 TO 10
120 r=RND(5)
130 PRINT "What city is the capital"
140 PRINT "of "; country$(r)
150 INPUT answer$
160 IF answer$=city$(r) THEN PROCyes ELSE PROCno
170 NEXT x
180 PRINT "You got ";right;
190 PRINT " correct out of 10"
200 END
500 DATA Paris, France, Reykjavik
505 DATA Iceland
510 DATA Moscow, Soviet Union
520 DATA Athens, Greece
530 DATA Spitzbergen, Spitzbergen
600 DEF PROCyes
610 PRINT "Well done!"
620 right=right+1
630 ENDPROC
700 DEF PROCno
710 PRINT "No, the capital of
720 PRINT country$(r);" is ";city$(r)
730 ENDPROC
```

Line 10 is just a REMark which the computer ignores.

Lines 20 and 30 tell the computer that we are going to use two string arrays – one to store the names of the five cities and the other to store the names of the five countries. See chapter 21 for an explanation of arrays.

Line 40 sets up a FOR...NEXT loop that will go around five times.

Line 50 reads the next word (which will be a city) into the array `city$` and then moves the 'data pointer' on to point to the next word (which will be a country).

Line 60 reads the next piece of `DATA` into the `country$` array.

Line 70 is the end marker of the `FOR...NEXT` loop.

Lines 110 to 170 loop ten times through a 'question and answer' quiz.

Lines 500 to 530 contain the `DATA` used above.

Lines 600 to 630 are a procedure to deal with correct replies.

Lines 700 to 730 deal with incorrect replies.

DEF <small>define</small>

Purpose

The word **DEF** is used to inform the computer that a procedure or function is about to be defined. Once the computer has been informed that this procedure or function exists, then the procedure or function can be called by name anywhere in the program.

Definitions of procedures and functions *must not occur in the body of a program*. They should be placed in a separate section *which is not executed* – for example after the final **END** in the program. This also aids readability.

The language BASIC has many predefined functions which the computer already knows about. For example, the function **SQR** enables it to calculate the square root of a number.

Often though, it is useful to be able to define your own functions. For example, you might want to have a function which calculates the VAT inclusive price of a product from the basic sale price by multiplying by 1.15.

A function always produces a result so you can write **X=FNST**. A procedure, on the other hand, is used to perform a number of actions, but it does not by itself produce a numerical result. For example, a procedure might be set up to clear the screen and draw a number of lines on the screen.

You may well feel confused, but do not be put off! The use of procedures and functions may be difficult to understand at first but it is well worth the effort. Their use greatly enhances the readability and reliability of programs.

The section below gives a more detailed explanation of the use of procedures and functions. It should be read in conjunction with the examples which follow.

Both procedures and functions may contain local variables which are declared using the word **LOCAL**. In the third example given below, K is declared as a local variable. This means that although K is used in this procedure its value is notdefined when the procedure finishes. In fact the variable K might well be used elsewhere in the program. The variable K, elsewhere within the program, would not be altered by the use of the local variable K within the procedure. Any variable which is not declared as **LOCAL** will be available outside the procedure, in other words to the rest of the program.

Also, both procedures and functions may have parameters passed to them. Look at the first example program below: line 1010 says

```
1010 DEF FNST(g)=1.15*g
```

'g' is called a 'formal parameter' for the fuction **FNST**. It tells the computer that one number is going to be passed to the function when the function is used and inside the function we have decided to use the letter g to represent the variable.

The procedure is 'called' or used like this – for example

```
230 PRINT "VAT inclusive price ";
236 PRINT FNST(P)
```

and in this case 'P' is the 'actual parameter' for the function **FNST**. Whatever value 'P' has will be used inside the fuction wherever reference is made to teh formal parameter 'g'. This is very convenient since you can use any variable names you like for the parameters inside the procedure. Then you can call the procedure with a quite different set of parameter names from the outside. Very often a procedure will be called from many different places in the program – and the actual parameters may have different names each time the procedure or function is called.

If a procedure or function is defined with (say) three formal parameters then, when it is called, three actual parameters must be supplied. See the fifth example below where three parameters are passed to the function.

The end of the procedure is indicated with the statement **ENDPROC**. The end of a multi-line function is indicated by the statement that starts with an = sign. The function is given the value of the expression to the right of the = sign.

Examples

First example – full program

```
 210 REPEAT
 220 INPUT "Basic price ",P
 230 PRINT "VAT inclusive price ";
 235 PRINT FNST(P)
 240 UNTIL P=0
 250 END
1000 REM line numeric function
1010 DEF FNST(g)=1.15*g
```

Second example – program section

Multi-line string function with one string parameter.

```
1000 DEF FNREVERSE(A$)
1010 REM reverse the order of the letters in A$
1015 REM
1020 LOCAL d%,B$
1030 FOR d%=1 TO LEN(A$)
```

```
1040 B$=MID$(A$,d%,1)+B$
1050 NEXT d%
1060 =B$
```

Third example – program section

Multi-line procedure with one parameter.

```
 200 DEF PROCbye(X)
 210 REM print bye X times
 220 LOCAL K
 230 FOR K=1 TO X
 240 PRINT "bye"
 250 NEXT K
 260 ENDPROC
```

Fourth example – program section

This sets the background colour to a new value given in the parameter.

```
 10 DEF PROCINITSCREEN(X)
 20 REM clear screen and draw border
 25 COLOR 128+X
 30 CLS
 40 DRAW 1279,0
 50 DRAW 1279,799
 60 DRAW 0,799
 70 DRAW 0,0
 80 ENDPROC
```

Fifth example - full program

```
 110 INPUT X,Y,Z
 120 M=FNMEAN(X,Y,Z)
 130 PRINT "The mean of ",X,Y,Z
 140 PRINT "is ";M
 150 END
8990 REM Single line numeric function
8995 REM with three parameters
9000 DEF FNMEAN(A,B,C)=(A+B+C)/3
```

Description

A program object which must precede declaration of a user function or procedure. String and numeric functions and procedures may be defined. Multi-line functions and procedures are allowed. All procedures and functions must be placed in the program where they will not be executed, eg after the **END** statement.

Syntax

DEF FN|PROC<variable name>[(<string>|<numeric>
{**,**<string>|<numeric>})]

Associated keywords

ENDPROC, FN, PROC

DEG degrees

Purpose

This function converts angles which are expressed in radians into degrees. A radian is equal to about 57 degrees.

Examples

```
100 X=DEG(PI/2)
```

```
300 angle=DEG (1.36)
```

```
PRINT DEG(PI/2)
```

Syntax

<num-var>=DEG<numeric>

Description

A function which converts radians to degrees.

Associated keywords

RAD, SIN, COS, TAN, ACS, ASN, ATN

DELETE

Purpose

The **DELETE** command is used to delete a group of lines from a program. It cannot be used as part of a program. You can specify which lines should be deleted with a command of the form

```
DELETE 120,340
```

This would remove everything between line 120 and line 340 inclusive.

To delete everything up to a certain line number (for example up to line 290) use **DELETE 0,290**.

To delete from line 500 to the last line, use as the last line to be deleted any number greater than the last line number in the program. Since the largest line number allowed is 32767

```
DELETE 500,32767
```

will do the trick, but will take a long time.

To delete a single line just type the line number and press **RETURN**. There is no need to use the **DELETE** command.

Examples

```
DELETE 0,540
```

```
DELETE 180,753
```

```
DELETE 540,32000
```

Syntax

DELETE<num-const>,<num-const>

Description

A command enabling a range of lines to be deleted from a program. Since **DELETE** is a command it cannot be used in a program or as part of a multiple statement line.

Associated keywords

LIST, **OLD**, **NEW**

DIM dimension of an array

Purpose

As well as simple numeric and string variables (such as 'X' and 'name$') it is possible to work with 'arrays' of variables. These are extremely useful when working with groups of numbers or words. For example if you wanted to work with a set of information about the rooms in a hotel with four floors, each with 30 rooms, then an array of four by 230 entries can be created like this:

```
DIM hotel(4,30)
```

Having set up an array, one can enter information into each of its 'elements'. For example the cost of the room per night might be £26.50

```
hotel(1,22)=26.50
hotel(4,1)=165.00
```

In practice the statement `DIM hotel(4,30)` produces an array of five by 31 entries since the lowest array element is `hotel(0,0)`.

All the above arrays are called 'two dimension numeric arrays'. Another array could contain the names of guests:

```
DIM name$(4,30)
name$(1,22)="Fred Smith"
name$(4,1)="The Queen"
```

That sort of array is called a 'two dimension string array'.

Arrays may have one or more dimensions. A single dimension array would be appropriate for all the houses in a road, eg

```
DIM MainSt(150)
```

That sort of array is called a 'single dimension numeric array'. All arrays are normally dimensioned very early in the program. It is 'illegal' to attempt to change the size of an array by re-dimensioning it later in the program. An array may have as many dimensions and as many elements in each dimension as the computer has space for – but you tend to run out of computer memory pretty fast with large arrays! It is essential that there is no space between the array name and the first parenthesis. Thus `DIM A(10)` is correct but `DIM A (10)` will not define an array.

Examples

```
100 DIM partnumbers(1000)

3000 DIM employeename$(35)

240 DIM ALL_hours_in_the_week(24,7)

100 DIM A(X)
```

Description

A statement which dimensions arrays. Arrays must be predeclared before use. After dimensioning all elements of arrays are initialised to zero for numeric arrays or null strings in the case of string arrays. The lowest element in an array is element zero. Thus `DIMX(4)` would create an array of five elements (0 to 4 inclusive).

There is a second and quite different use for the `DIM` statement. It can be used to reserve bytes in memory for special applications. To reserve 25 bytes, type

```
DIM X 24
```

Notice two things about this statement: firstly the space between the variable X and the (number of bytes minus 1) and secondly the absence of parentheses around the `24`. The address of the start of the group of 25 bytes is given in the variable `X` in this example.

Syntax

`DIM`<num-var>|<str-var>(<numeric>{,<numeric>})

or

`DIM`<num-var> <numeric>

Associated keywords

None

DIV division of whole numbers

Purpose

See the keyword `MOD` for a full explanation. `DIV` is an operator which gives the whole number part of the result of a division. Thus

```
PRINT 11 DIV 4
```

gives 2 (leaving a remainder of 3).

Description

A binary operator performing integer division between its operands. The operands are truncated to integers before division takes place.

Syntax

<num-var>=<numeric>DIV<numeric>

Associated keywords

`MOD`

DRAW

Purpose

This statement draws lines on the screen in **MODE**s **0**, **1**, **2**, **4** and **5**. The **DRAW** statement is followed by two numbers which are the X and Y coordinates of the end of the line. The line starting point can either be the end of the last line that was drawn or else a new point if the **MOVE** statement has been used before the statement **DRAW**.

The screen is addressed as

1280 points wide X-axis, 0-1279
1024 points high Y-axis, 0-1023

regardless of the graphics **MODE** selected. The origin (position 0,0) is normally at the bottom left of the screen.

The line is drawn in the current graphics foreground colour. This can be changed using the **GCOL** statement.

Examples

```
780 DRAW X,Y
```

```
DRAW 135,200
```

Description

DRAW X,Y means draw a line to **X,Y** in the current foreground colour.

DRAW X,Y is equivalent to **PLOT 5,X,Y**.

DRAW is one of a large group of line drawing statements. See **PLOT** for others.

Syntax

DRAW<numeric>**,**<numeric>

Associated keywords

MODE, PLOT, MOVE, CLG, VDU, GCOL

Demonstration program

```
140 MODE 5
160 REM Red background
170 GCOL 0, 129
175 CLG
180 REM Yellow foreground
190 GCOL 0,2
200 REM draw a box
210 MOVE 100,100
220 DRAW 400,100
230 DRAW 400,400
240 DRAW 100,400
250 DRAW 100,100
```

ELSE

Purpose

To provide an alternative course of action. **ELSE** can be used following an **IF...THEN** statement, an **ON...GOTO** statement, and an **ON...GOSUB** statement. See the pages describing the associated keywords and chapter 16 for more details.

Examples

```
560 IF length > 0 THEN PRINT "OK" ELSE PRINT "No
good"
```

```
100 IF A<>B THEN C=D ELSE PRINT "Values match"
```

Description

Part of the **IF...THEN...ELSE** structure.

Syntax

IF<testable condition>[**THEN**]<statement>[**ELSE**<statement>]

or

ON<num-var>**GOTO**<numeric>{**,**<numeric>}[**ELSE**<statement>]

or

ON<num-var>**GOSUB**<numeric>{**,**<numeric>}[**ELSE**<statement>]

Associated keywords

IF, **THEN**, **ON**

END

Purpose

This informs the computer that it has reached the end of the program. `END` is optional but may be used as many times as required in a program.

Example

```
9000 END
```

Description

Optional end of program which may occur anywhere and as often as is required.

The command `END` has a special use in that it causes BASIC to search the program in memory for a valid end program marker. BASIC then updates its internal pointers. This may be useful after unusual loading procedures. If the user changes the value of `PAGE` then internal pointers such as `TOP` will not be reset until an `END` statement or command is met.

Syntax

```
END
```

Associated keywords

```
STOP
```

ENDPROC end procedure

Purpose

This indicates the end of a procedure definition. See the keyword **DEF** for more information.

Examples

```
1000 DEF PROCdash(param)
1010 REM print dashes lots of times
1020 REM in fact "param" dashes in total
1025 REM
1030 LOCAL counter
1040 FOR counter=1 TO param
1050 PRINT"-";
1060 NEXT counter
1070 ENDPROC

2010 DEF PROCtriangle(A,B,C,D,E,F)
2020 REM fill a triangle with colour
2050 MOVE A,B
2060 MOVE C,D
2070 PLOT 85,E,F
2100 ENDPROC
```

Description

Part of the **DEF PROC...ENDPROC** structure.

Syntax

ENDPROC

Associated keywords

DEF, FN, PROC, LOCAL

ENVELOPE

Purpose

The **ENVELOPE** statement is used with the **SOUND** statement to control the volume and pitch of a sound while it is playing. All natural sounds change in volume (loudness or amplitude); for example, the sounds from a piano start off loudly and then fade away. An aircraft flying overhead starts off softly, gets louder and then fades away.

The variation of amplitude (loudness) for the aircraft, as it flies overhead, looks something like this:

This variation of amplitude with time is described as an 'amplitude envelope'.

Some sounds change in pitch. For example, a wailing police siren:

This variation of pitch with time is called a 'pitch envelope'.

The BBC Microcomputer can use both pitch and amplitude envelopes and these are set up with the **ENVELOPE** statement.

Example

```
10 ENVELOPE 1,1,4,-4,4,10,20,10,127,0,0,-5,126,126
20 SOUND 1,1,100,200
```

Description

The **ENVELOPE** statement is followed by 14 parameters.

```
ENVELOPE
N,T,PI1,P12,PI3,PN1,PN2,PN3,AA,AD,AS,AR,ALA,ALD
```

Parameter		Range	Function
N		1 to 4	Envelope number
T	bits 0-6	0 to 127	Length of each step in hundredths of a second
	bit 7	0 or 1	0 = auto-repeat pitch envelope
			1 = don't auto-repeat pitch envelope
PI1		-128 to 127	Change of pitch per step in section 1
PI2		-128 to 127	Change of pitch per step in section 2
PI3		-128 to 127	Change of pitch per step in section 3
PN1		0 to 255	Number of steps in section 1
PN2		0 to 255	Number of steps in section 2
PN3		0 to 255	Number of steps in section 3
AA		-127 to 127	Change of amplitude per step during attack phase
AD		-127 to 127	Change of amplitude per step during decay phase
AS		-127 to 0	Change of amplitude per step during sustain phase
AR		-127 to 0	Change of amplitude per step during release phase
ALA		0 to 126	Target level at end of attack phase
ALD		0 to 126	Target level at end of decay phase

The N parameter specifies the envelope number that is to be defined. It normally has a value in the range 1 to 4. If the BASIC statement **BPUT#** is not being used then envelope numbers up to and including 16 may be used.

The T parameter determines the length in hundredths of a second of each step of the pitch and amplitude envelopes. The pitch envelope normally auto-repeats but this can be suppressed by setting the top bit of T – ie using values of T greater than 127.

The six parameters PI1, P12, P13, PN1, PN2 and PN3 determine the pitch envelope. The pitch envelope has three sections and each section is specified with two parameters: the increment, which may be positive or negative, and the number of times the increment is to be applied during that section, that is the number of steps. A typical pitch envelope might look like

In the above example T = one hundredth of a second.

PI1 = +10	PN1 = 12
PI2 = -5	PN2 = 27
PI3 = +50	PN3 = 3

The pitch envelope is added to the pitch parameter (P) given in the **SOUND** statement. In the above example it must have been 40 since the pitch starts at 40. If bit 7 of the T parameter is zero then at the end of the pitch envelope, at a time given by the equation

time = (PN1 + PN2 + PN3)*T hundredths of a second

the pitch envelope will be set to zero and will repeat automatically. Note that the pitch can only take on values in the range 0 to 255 and values outside this range are treated as **MOD 256** of the value calculated.

The six parameters AA, AD, AS, AR, ALA and ALD determine the amplitude envelope. Although the current internal sound generator has only 16 amplitude levels the software is upward compatible with a generator having 128 levels.

The shape of the amplitude envelope is defined in terms of rates (increments) between levels, and is an extended form of the standard ADSR system of envelope control. The envelope starts at zero and then climbs at a rate set by AA (the attack rate) until it reaches the level set by ALA. It then climbs or falls at the rate set by AD (the decay rate) until it reaches the level set by ALD. However, if AD is zero the amplitude will stay at the level set by ALA for the duration (D) of the sound.

The envelope then enters the sustain phase which lasts for the remaining duration (D) of the sound. The duration is set by the **SOUND** statement. During the sustain phase the amplitude will remain the same or fall at a rate set by AS.

At the end of the sustain phase the note will be terminated if there is another note waiting to be played on the selected channel. If no note is waiting then the amplitude will decay at a rate set by AR until the amplitude reaches zero. If AR is zero then the note will continue indefinitely, with the pitch envelope auto-repeating if bit 7 of parameter T is zero.

A typical amplitude envelope might look like

In the above example T = one hundredth of a second.

ALA = 120
ADA = 80
AA = 30 (120 in four hundredths of a second)
AD = -4 (-40 in 10 hundredths of a second)
AS = 0
AR = -5 (-80 in 16 hundredths of a second)

Note that the amplitude cannot be moved outside the range 0 to 126.

Syntax

ENVELOPE <numeric>,<numeric>,<numeric>,<numeric>,
<numeric>,<numeric>,<numeric>,<numeric>,<numeric>,
<numeric>,<numeric>,<numeric>,<numeric>,<numeric>

Associated keywords

ADVAL, **SOUND**

EOF#

Purpose

This function is used to determine whether the end of the file has been reached or not. The function returns the value 0 or -1. It returns the value -1 if the end of the file has been reached. The number following **EOF#** is the channel number of the file. Refer to chapter 31 for more information.

Examples

```
100 X=EOF# (channel)
```

```
200 REPEAT UNTIL EOF#(y)
```

Description

The function used to determine whether the end of the file has been reached or not.

Syntax

<num-var>=**EOF#**(<num-var>)

Associated keywords

OPENIN, OPENUP, OPENOUT, EXT#, PTR#, PRINT#, INPUT#, BGET#, BPUT#, CLOSE#

EOR Exclusive-OR

Purpose

This is a special logical operator often used to complement certain bits in a byte selectively. Refer to the keyword **AND** for an introduction to the concepts involved.

The process of Exclusive-OR tests whether the corresponding bits in two numbers are the same or different. If the corresponding bits in the two numbers are different then the resultant bit will be a one, if they are the same it will be a zero.

Another way of looking at this process is that it complements (changes 0 to 1 and 1 to 0) those bits in one number which are at logic 1 in the other number. Thus if

X is	0000 1100 0011 0000 1110 1011
Y is	1011 1111 0000 1010 0010 1000

Then

X **EOR** Y is	1011 0011 0011 1010 1100 0011

Examples

```
100 d%= A% EOR &FFFF00
200 R= X EOR Y
```

Description

An operator performing the operation of logical or bitwise Exclusive-OR between the two operands.

Syntax

<num-var>=<numeric>**EOR**<numeric>

or

<testable condition>=<testable condition>**EOR**<testable condition>

Associated keywords

NOT, AND, OR

ERL error line number

Purpose

This enables the program to find out the line number where the last error occurred. See chapter 27 for more information.

Examples

```
8500 X=ERL

8100 IF ERL=100 THEN PRINT "I didn't understand"

300 IF ERL=10000 THEN CLOSE#0
```

Description

A function returning the line number of the line where the last error occurred.

Syntax

<num-var>=**ERL**

Associated keywords

ON ERROR..., **ON ERROR OFF**, **REPORT**, **ERR**

ERR error

Purpose

If the computer finds an error that it cannot resolve, it may give up and report the error on the screen. In addition it remembers an 'error number'. For example, if you try to calculate with numbers which are too large for the computer it will report **Too big** and remember error number 20.

Pressing the **ESCAPE** key behaves as an error (error number 17) and you can detect this and act on it if you wish.

It is possible to make the computer deal with most of these errors itself by writing special sections of the program to deal with the inevitable! These sections of the program need to know what the error was and where it occurred.

The function **ERR** enables your program to find the error number of the last error which occurred. This is generally used to enable the program to respond helpfully to an error caused by the user.

Examples

```
1000 wrong=ERR

100 IF ERR=17 THEN PRINT "YOU CAN'T ESCAPE!"

1230 IF ERR=18 THEN PRINT "You can't divide by
zero!"
```

Description

Returns the error number of the last error which occurred.

Syntax

<num-var>=**ERR**

Associated keywords

ON ERROR..., **ON ERROR OFF**, **ERL**, **REPORT**

EVAL evaluate

Purpose

This function is mainly used to enable the user to type an expression, such as a mathematical equation, into the computer while a program is running.

For example, suppose that a program has to plot a graph; you need a way of getting your equation into the computer while a program is running. In most versions of BASIC this is very difficult to do. With the BBC BASIC the equation is put into a string and then **EVAL** is used to tell the computer to work out the string.

The function is not common in other versions of BASIC so a few more specific examples are given of instructions which can be evaluated by the statement **EVAL A$**.

```
A$="M*X+C"
```

```
A$="SIN(x/120)+COS(x/30)"
```

Note that **EVAL** can be used to evaluate functions (**SIN**, **COS**, **SQR** etc) but cannot be used to execute a statement like **MODE 4**.

Examples

```
100 X=EVAL(A$)
```

```
234 value=EVAL (z$)
```

Description

A statement which applies the interpreter's expression evaluation program to the characters held in the argument string. An easy way to pass a function into a program from a user input.

Syntax

<num-var>=**EVAL**(<string>)

or

<str-var>=**EVAL**(<string>)

Associated keywords

VAL, **STR$**

Demonstration program

```
10 INPUT A$
20 FOR X=1 TO 5
30 Y=EVAL(A$)
40 PRINT Y
50 NEXT X
>RUN
?5*X
         5
        10
        15
        20
        25
```

This second program makes the computer act as a calculator.

```
 5 REPEAT
10 INPUT B$
20 PRINT EVAL B$
30 UNTIL FALSE
>RUN
?3+4
         7
?SIN (RAD (45))
0.707106781
?
```

EXP exponent

Purpose

This mathematical function calculates *e* (2.7183…) raised to any specified power.

Examples

```
120 Y=EXP(X)
3000 pressure=EXP(height)
```

Description

A function returning *e* to the power of its argument.

Syntax

<num-var>=EXP(<numeric>)

Associated keywords

LN, LOG

EXT# extent

Purpose

This function finds out how large a particular file is. It only works with disc and Econet filing systems – not with cassette files. The number returned is the number of bytes allocated to the file. The file to be investigated must have been opened during the **OPENIN**, **OPENOUT** or **OPENUP** statements. See chapter 31 for more information on file handling.

Examples

```
100 X=EXT#(employee)
```

```
PRINT EXT# (N)
```

Description

A function which returns the length in bytes of the file opened on the channel given in its argument.

Syntax

<num-var>=**EXT#**(<num-var>)

Associated keywords

CLOSE#, **PTR#**, **INPUT#**, **PRINT#**, **BGET#**, **BPUT#**, **OPENIN**, **OPENUP**, **OPENOUT**, **EOF#**

FALSE

Purpose

Sometimes the computer has to decide whether or not something is true. For example:

```
10 X=12
20 IF X=20 THEN PRINT "X EQUALS 20"
```

Clearly, in this example, the statement `x=20` is false. As program will never print `X EQUALS 20`.

```
100 X=12
110 REPEAT
120 PRINT "HELLO"
130 UNTIL X=20
```

would repeatedly print `HELLO` because X will never be anything other than 12. `X=20` is `FALSE`. The same effect can be achieved by writing

```
110 REPEAT
120 PRINT "HELLO"
130 UNTIL FALSE
```

which means repeat for ever.

In fact, the computer has a numerical value of `FALSE`, which is zero. Thus

`PRINT FALSE`

will print `0`.

Similarly

`PRINT 5=4`

will also print `0`, since `5=4` is false.

It is often useful to say in a program for example

`CLOCKSET = FALSE`

and then later you can say

`IF CLOCKSET THEN PRINT "THE CLOCK IS CORRECT"`

Examples

```
100 oldenough = FALSE
```

```
245 UNTIL FALSE
```

Description

A function returning the value zero.

Syntax

<num-var>=**FALSE**

Associated keywords

TRUE

FN function

Purpose

FN preceding a variable name indicates that it is being used as the name of a function. Both string and numeric functions may be defined. See the keyword **DEF** and chapters 17 and 18 for a more detailed description of functions and procedures.

Since a function always returns a value. It often appears on the right of an equal sign or in a print statement. Procedures, on the other hand, do not return results.

Example

```
1000 DEF FNmean2(x,y)=(x+y)/2
```

Description

A reserved word used at the start of all user defined functions.

Syntax

DEF FN<variable-name>[(<num-var>|<str-var>{,<num-var>|<str-var>})]

Associated keywords

ENDPROC, **DEF**, **LOCAL**

FOR

Purpose

The word **FOR** is one of the words used in a **FOR...NEXT** loop. This makes the computer execute a series of statements a specified number of times; for example

```
120 FOR X=1 TO 5
130 PRINT X
140 NEXT X
```

would print out the numbers **1**, **2**, **3**, **4**, **5**.

The variable **X** in the above example initially takes on the value 1 and the program then goes through until it reaches the word **NEXT**. The program then returns to the line or statement **FOR X=1 TO 5** and **X** is increased in value by 1. The program continues the loop, increasing the value of **X** in steps of 1, until **X** reaches 5. After that, the program no longer loops; instead it moves onto the next statement after the **FOR...NEXT** loop.

As an option the 'step size' can be changed. In the example above **X** increased by 1 each time around the loop. In the next example **XYZ** increases by 0.3 each time around the loop.

```
230 FOR XYZ=5 TO 6 STEP 0.3
240 PRINT XYZ
250 NEXT XYZ
```

The above program would print out the numbers

```
    5
   5.3
   5.6
   5.9
```

The value of **XYZ** on exit from the above program would be 6.2.

The step size may be negative if you wish to make the value of the control variable decrease each time around the loop.

```
870 FOR r2d2%=99 TO 60 STEP -12
880 PRINT r2d2%; "Hi there"
890 NEXT r2d2%
```

would print

```
        99Hi there
        87Hi there
        75Hi there
        63Hi there
```

The `FOR...NEXT` loop always executes once so

`FOR D=5 TO 3: PRINT D: NEXT D`

would print **5** and then stop.

Examples

`300 FOR X=1 TO 16 STEP 0.3: PRINT X: NEXT X`

`1040 FOR A%=0 TO MAXIMUM%`

`560 FOR TEMPERATURE=0 TO 9`

Description

A statement initialising a `FOR...NEXT` loop. This structure always executes at least once. Any assignable numeric item may be used as the control variable but integer control variables are about three times faster than real variables. Note that when the step is non-integer, rounding errors may creep in and the value of the control variable may well diverge significantly from the true arithmetic value.

Syntax

`FOR` <num-var>=<numeric>`TO`<numeric>[`STEP`<numeric>]

Associated keywords

`TO`, `STEP`, `NEXT`

GCOL

Purpose

This statement sets the colour to be used by all subsequent graphics operations. In other words it selects the graphics foreground colour and the graphics background colour. It also specifies how the colour is to be placed on the screen. The colour can be plotted directly, ANDed, ORed or Exclusive-ORed with the colour already there, or the colour there can be inverted (reversed).

The first number specifies the mode of action as follows:

0	Plot the colour specified
1	OR the specified colour with that already there
2	AND the specified colour with that already there
3	Exclusive-OR the specified colour with that already there
4	Invert the colour already there

The second number defines the logical colour to be used in future. If the number is greater than 127 then it defines the graphics background colour. If the number is less than 128 then it defines the graphics foreground colour. See the keyword **COLOUR** for more information.

Examples

```
100 GCOL 0,2
```

```
GCOL 3,129
```

Description

This statement is used to select the logical colours used by graphics statements.

Syntax

GCOL<numeric>,<numeric>

Associated keywords

CLS, CLG, MODE, COLOUR, PLOT

GET

Purpose

This function waits for a key to be pressed on the keyboard and then returns with the ASCII number of the key pressed. See chapter 9 for a description of ASCII numbers.

The `GET` function is used whenever the computer needs to wait for a reply from the user before continuing.

Note that when using `GET` the character typed on the keyboard will not appear on the screen. If you wish it to appear you must then ask the computer to print it.

Examples

```
1040 keyhit=GET
```

```
350 X=GET
```

Description

A function which waits for the next character from the input stream. The function then returns the ASCII value of the character.

Syntax

<num-var>=`GET`

Associated keywords

`GET$`, `INKEY`, `INKEY$`

GET$

Purpose

The `GET$` function waits for a key to be pressed on the keyboard and then returns with a string containing the character pressed. See the previous keyword, `GET`, for a similar function and for further explanation.

For example, at the end of a game program you may wish the computer to ask the player whether or not he or she wants to go again. The demonstration program below shows how this can be done.

Note that when using `GET$` the character typed on the keyboard will not appear on the screen. If you wish it to appear you must ask the computer to print it.

Example

```
1050 A$=GET$
```

Syntax

<string-var>=GET$

Associated keywords

`GET`, `INKEY`, `INKEY$`

Demonstration program

```
PRINT "Do you want to play another game";
2120 REM if user presses Y then
2130 REM go to line 100
2140 IF GET$="Y" THEN GOTO 100
2160 REM if it gets this far then the
2170 REM reply was not "Y" so give up!
2180 STOP
```

GOSUB go to a subroutine

Purpose

Often a group of lines in a program needs to be used in a number of different places within the main program. Instead of repeating the same piece of program several times you can make this small sub-section into a separate subroutine. This subroutine can then be 'called' from a number of different places in the main program by means of the statement **GOSUB**. The end of a subroutine is indicated by the word **RETURN**. This causes the program to return to the statement after the **GOSUB** statement.

Beware of a subroutine calling itself too many times: a depth of 26 subroutines is the maximum that is allowed.

As with **GOTO**, it is possible to **GOSUB** to a calculated line number. The same cautions that apply to **GOTO** apply to **GOSUB** in this case.

Example

```
1020 GOSUB 4000
```

Description

A statement used to call a section of program as a subroutine. One subroutine may call another subroutine (or itself) up to a maximum nested depth of 26.

Syntax

GOSUB<numeric>

ON<num-var>GOSUB<numeric>{,<numeric>}[ELSE<statement>]

Associated keywords

RETURN, ON

Demonstration program

First, here is a program to print out random phrases without using a subroutine:

```
100 REM A$ contains 7 words and each word
105 REM contains 5 characters - letters or spaces
110 A$="hand mouthear  leg  arm  chestelbow"
120 FOR count=1 TO 10
125 REM pick a random number
130 R=RND(7)
```

```
140 REM and use it to pick a random word
150 B$=MID$(A$,5*R-4,5)
160 REM print a message
170 PRINT "My ";B$;" hurts"
180 REM get another random word
190 R=RND(7)
200 B$=MID$(A$,5*R-4,5)
210 REM and print out a second message
220 PRINT "Is your ";B$;" all right?"
230 NEXT count
```

Now look at the same program using a subroutine and with the **REM**s (remarks) removed.

```
110 A$="hand mouthear  leg  arm  chestelbow"
120 FOR count=1 TO 10
150 GOSUB 810
170 PRINT "MY ";B$;" hurts"
190 GOSUB 810
220 PRINT "Is your ";B$;" all right?"
230 NEXT count
240 END
810 B$=MID$(A$,5*RND(7)-4,5)
820 RETURN
```

The 7 in line 810 is there to select one of the seven words in A$. In line 810 both the 5s are there because each word contains five letters or spaces. It is essential that all the words contain the same number of characters.

Finally, here is the same program written with a string function, with **REM**s left out and generally cleaned up.

```
110 A$="hand mouthear  leg  arm  chestelbow"
120 FOR count=1 TO 10
170 PRINT "My ";FNword;" hurts"
220 PRINT "Is your ";FNword;" all right?"
230 NEXT count
240 END
800 DEF FNword= MID$(A$,5*RND(7)-4,5)
```

GOTO go to a line number

Purpose

This statement makes the computer jump to a specified line number instead of continuing to the next one in the program. It changes the order in which the computer executes a program.

Although **GOTO** is simple to use, do so with caution! It is all too easy to make a program difficult to follow by using too many **GOTO**s. Following a program full of **GOTO**s is like trying to disentangle a plateful of spaghetti and arrange it in a straight line!

Adherents of 'structured programming' encourage program writers to use structures like **REPEAT...UNTIL** and **FOR...NEXT** and to avoid most (but not all) **GOTO** statements.

It is possible in this version of BASIC to **GOTO** a variable. In the following example the destination variable is called 'somewhere':

```
10 somewhere=1005
20 GOTO somewhere
```

but this feature must be used with great care since the program cannot be renumbered using the **RENUMBER** command.

Note that if the destination line number is to be calculated using a mathematical expression then that expression must be in parentheses.

GOTO can be used as a command to start a program without destroying the values assigned to the variables.

Examples

```
GOTO 330

100 IF X>5 THEN GOTO 2000

100GOTO (starts*55+14)
```

Description

A statement used to transfer control to a specified or calculated line number unconditionally.

Syntax

`GOTO`<numeric>

`ON`<num-var>`GOTO`<numeric>`{,`<numeric>`}[``ELSE`<statement>]

HIMEM highest memory location

Purpose

BASIC uses the computer's random access memory (RAM) to store the user's program, all the variables that the program uses, and memory for high-resolution graphics displays.

In the absence of other instructions the computer divides the available memory up logically. However there are occasions, particularly when changing display modes and when writing machine code programs, when you may wish to tell BASIC how to divide up the available memory.

One way of changing the allocation is by altering the value of the variable `HIMEM`. This variable contains the address of the highest memory location that BASIC uses for your program and variables. It is automatically set to just below the memory used for the screen when the `MODE` is selected. Addresses above `HIMEM` are not used by BASIC.

If it is manually altered then locations above `HIMEM` may be used by the programmer for other things, for example for machine code subroutines.

If you wish to change the value of `HIMEM` you should normally do so very early in your program – preferably right at the beginning. The beginning of the program is also the place to select the display mode that you will be using.

Other important boundaries are `PAGE`, `TOP` and `LOMEM`. The memory map in Appendix J gives an indication of their relative positions.

Note that in the 'shadow screen' mode, `HIMEM` always returns a value of &8000 (see chapter 42 for more details).

Examples

```
100 HIMEM=HIMEM-40
```

```
100 PRINT HIMEM
```

```
100 HIMEM = &2800
```

Description

`HIMEM` contains the address of the first byte that BASIC does not use. This pseudo-variable must not be altered while executing a function or a procedure. Alter it with great care! When using a second processor, or if the computer is being used in shadow mode, `HIMEM` will not be altered when changing `MODE`.

Syntax

`HIMEM=<numeric>`

or

`<num-var>=HIMEM`

Associated keywords

`LOMEM`, `PAGE`, `TOP`

IF

Purpose

This sets up a test condition which can be used to control the subsequent action of the computer.

Examples

```
100 IF month=12 THEN PRINT "December"

100 IF A=1 THEN PRINT "One" ELSE PRINT "Not one"

100 IF answer$="BANANA" THEN PROCfruit

100 IF height<1.94 OR age<18 THEN GOTO 1030

100 IF length <>5 THEN 2140

100 IF RATE=5 THEN Y=6:Z=8 ELSE PRINT "Wrong rate"

100 IF month=11 THEN IF day=5 THEN PRINT "Guy
Fawkes"

100 IF month=1 AND day=1 THEN PRINT "New Year"

100 IF X THEN Y=0
```

Description

A statement forming part of the **IF...THEN...ELSE** structure. The word **THEN** is optional as is the **ELSE** section.

Syntax

IF<testable
condition>[**THEN**]<statement | numeric>[**ELSE**<statement | numeric>]

Associated keywords

THEN, ELSE

INKEY input the number of the key pressed

Purpose

This function waits for a specified time while constantly testing to see if a key has been pressed on the keyboard.

If a key is pressed before the time runs out then the ASCII value of the key is given. If no key is pressed in the given time then -1 is returned and the program continues. See the keyword **ASC** for an explanation of ASCII values.

Note that a key can be pressed at any time before **INKEY** is used. All keys pressed are stored in a buffer in the computer and a character is removed from the buffer by, for example, the **INPUT** statement. You can clear the buffer of all characters by giving the command

```
*FX 15,1
```

The number in parenthesis, after the word **INKEY**, gives the amount of time that the computer must wait before giving up. The time is given in hundredths of a second, and may have any value between 0 and 32767.

In addition, the function **INKEY** can be used to see if a key is actually pressed at the instant the function is called. Normally pressing a key once enters the code for that key into the keyboard buffer. If the key is kept down then it will normally auto-repeat and further characters will be entered into the buffer. However, when the buffer is read with **INPUT** or **GET** or **INKEY**, you will have no idea how long the character has been waiting in the buffer. An alternative statement is provided which actually tests the keyboard rather than the buffer.

INKEY with a negative number in the parenthesis, eg **INKEY(-27)** will enable you to test to see whether a particular key is pressed at that instant. The number in parenthesis determines which key you wish to test. The following table shows the negative number to be used to test any particular key. Thus the letter L would be tested with **PRINT INKEY (-87)**.

Examples

```
100 keynumber=INKEY(5)
```

```
220 result=INKEY(Y)
```

```
X=INKEY(100)
```

Description

A function which waits up to a specified time for a key to be pressed. The function returns -1 if no key is pressed in the specified time, or the ASCII value of the key pressed. The argument is the maximum time in hundredths of a second.

Syntax

<num-var>=**INKEY**(<numeric>)

Associated keywords

GET, **GET$**, **INKEY$**

Key	Number	Key	Number
f0	-33	1	-49
f1	-114	2	-50
f2	-115	3	-18
f3	-116	4	-19
f4	-21	5	-20
f5	-117	6	-53
f6	-118	7	-37
f7	-23	8	-22
f8	-119	9	-39
f9	-120	0	-40
A	-66	-	-24
B	-101	^	-25
C	-83	\	-121
D	-51	@	-72
E	-35	[	-57
F	-68	–	-41
G	-84	;	-88
H	-85	:	-73
I	-38	]	-89
J	-70	’	-103
K	-71	.	-104
L	-87	/	-105
S	-82	Space bar	-99
M	-102	**ESCAPE**	-113
N	-86	**TAB**	-97
O	-55	**CAPS LOCK**	-65
P	-56	**CTRL**	-2
Q	-17	**SHIFT LOCK**	-81
R	-52	**SHIFT**	-1
T	-36	**DELETE**	-90
U	-54	**COPY**	-106
V	-100	**RETURN**	-74
W	-34	↑	-58
X	-67	↓	-42
Y	-69	←	-26
Z	-98	→	-122

INKEY$ input the character pressed

Purpose

This function waits for a specified time while constantly testing to see if a key has been pressed on the keyboard. If a key is pressed before the time runs out then the letter or number pressed is placed in the string variable. If no key is pressed in the given time then an empty string is returned and the program continues.

Note that a key can be pressed at any time before **INKEY$** is used. All keys pressed are stored in a buffer in the computer and a character is removed from the buffer by, for example, the **INPUT** statement. You can clear the buffer of all characters by giving the command

```
*FX 15,1
```

The number in parenthesis, after the word **INKEY$**, gives the amount of time that the computer must wait before giving up. The time is given in hundredths of a second.

Examples

```
120 letter$=INKEY$(0)
```

```
384 result$=INKEY$(100)
```

```
920 X$=INKEY$(Y)
```

Description

A function which waits for a key to be pressed within a specified period of time. The function returns a null string if no key is pressed in the specified time. If a key is pressed the string returned consists of the single character pressed. The argument is the maximum time in hundredths of a second.

Syntax

<string-var>=**INKEY$**(<numeric>)

Associated keywords

GET, **GET$**, **INKEY**

INPUT to put information into the computer

Purpose

When a computer program is running there is often a need to get numbers or words from the outside world into the computer so that it can do calculations on these numbers or words. The statement **INPUT** is used for this purpose. There are a number of options:

```
100 INPUT X
```

will print a question mark on the screen and wait for the user to type in a number. This is not very 'friendly' – often it would be helpful to print a message on the screen before waiting for the user to type his/her reply. This can be done in two ways:

```
340 PRINT "How old are you";
350 INPUT AGE
```

or more simply

```
340 INPUT "How old are you",AGE
```

If you do not wish the computer automatically to print a question mark then omit the comma between the message to be printed out and the variable to be filled in.

```
340 INPUT "How old are you" AGE
```

Often you may want to input several values one after the other. This can be done by placing the variables after each other, but separated by commas, thus:

```
560 INPUT "Pick three numbers",X,Y,Z
```

When replying the user separates the values entered either with commas or by pressing the **RETURN** key after entering each value. The numbers that are typed in are placed in the appropriate variables – **X**, **Y** and **Z** in the example above.

The above examples all required numbers to be supplied by the user. You can **INPUT** words as well.

```
250 INPUT "What is your name", NAME$
```

You can **INPUT** more than one string at a time if you wish by using

```
200 INPUT "Town" ,A$,"Country",B$
```

INPUT LINE A$** will accept everything that is typed in including leading spaces and commas, and will place everything into **A$.

INPUT may be used with **TAB** in the same way that **PRINT** can be. For example

300 INPUT TAB(3,12) "number please" X

SPC can be used also, to insert spaces.

A semi-colon can replace a comma. Either a semi-colon or a comma will cause the computer to print a question mark when waiting for the variable to be filled.

Description

A statement to input values from the current input stream. The question mark prompt may be suppressed by omitting the comma following the prompt string. **INPUT** strips leading spaces off strings.

Syntax

INPUT[<string-const>][**;**|**,**]<num-var>|<string-var>**{,**<num-var>|<string-var>**}**

Associated keywords

INPUT#, **LINE**, **TAB**, **SPC**

INPUT# put information into the computer from

cassette or disc

Purpose

It is possible to record data (numbers and words) on cassette or disc where they can be stored for later use. The statement **INPUT#** is used to read the data back into the computer from the cassette or disc. See chapter 31 on file handling for more information.

Examples

```
1200 INPUT# channel, date, name$, address$
```

```
3400 INPUT#X,U,V,W$
```

Description

A statement which reads data in *internal format* from a file and places the data in the stated variables.

Syntax

INPUT#<num-var>,<num-var>|<string-var>{,<num-var>|<string-var>}

Associated keywords

OPENIN, **OPENUP**, **OPENOUT**, **EXT#**, **PTR#**, **PRINT#**, **BGET#**, **BPUT#**, **CLOSE#**

INSTR in string

Purpose

To search one string for any occurrence of another string, for example to see if one word contains another specific word.

The search normally starts from the beginning of one string but as an option the search can start from a specified point along the string.

The number returned is the string position of the second string in the first string. The leftmost character position is position number 1. If no match is found then zero is returned. A search for a null string

```
X=INSTR("Sunday","")
```

will always return 1.

Examples

```
240 X=INSTR(A$,B$)
```

puts the position of `B$` in `A$` into `X`.

```
180 Y=INSTR(A$,B$,Z)
```

starts search at position `Z`.

```
PRINT INSTR("HELLO","L")
```

would print `3`.

Description

A function which returns the position of a sub-string within a string. The starting position for the search may be specified. There must be no space between `INSTR` and the first parenthesis.

Syntax

<num-var>=INSTR(<string>,<string>[,<numeric>])

Associated keywords

`LEFT$`, `MID$`, `RIGHT$`, `LEN$`

INT integer part

Purpose

This converts a number with a decimal part to a whole number. This function always returns a whole number smaller than the number supplied. Thus `INT(23.789)` gives 23 whereas `INT(-13.3)` returns -14.

Examples

```
200 X = INT(Y)

1050 wholenumber=INT (decimalnumber)

330 pence=INT(cost * markup/quantity)
```

Description

`INT` is a function converting a real number to the lower integer.

Syntax

<num-var>=**INT**<numeric>

LEFT$ left string

Purpose

To copy part of a string starting at the left of the source string. For example if

```
A$="CATASTROPHE"
```

then

```
PRINT LEFT$(A$,3)
```

would give `CAT`, namely the left three characters of `A$`.

Examples

```
100 INDEX$=LEFT$(WHOLEname$,4)
```

```
3000 U$=LEFT$(H4$,value)
```

Description

A string function which returns the left n characters from a string. If the source string is too short then the function returns with as many characters as there are in the source string. There must be no space between `LEFT$` and the first parenthesis.

Syntax

<string-var>=`LEFT$`(<string>,<numeric>)

Associated keywords

`RIGHT$`, `MID$`, `LEN`, `INSTR`

Demonstration program

This prints out letters in a pattern.

```
10 PRINT "What is your full name";
20 INPUT name$
30 FOR X = 1 TO LEN(name$)
40 PRINT LEFT$(name$,X)
50 NEXT X
>RUN
```

```
What is your full name?JOHN A COLL
J
JO
JOH
JOHN
JOHN
JOHN A
JOHN A
JOHN A C
JOHN A CO
JOHN A COL
JOHN A COLL
```

LEN length (of a string)

Purpose

This function counts the number of characters in a string. For example

```
K=LEN("FRIDAY ")
```

would give `K=7` since there are six letters in `"FRIDAY"` and it is followed by a space.

This function is often used with a `FOR...NEXT` loop to do something once for each letter in a string. For example, we might wish to encode a word by replacing each letter with its successor in the alphabet so that, for example, `"FRIDAY"` would become `"GSJEBZ"`. See the demonstration program.

Examples

```
100 X=LEN(A$)
```

```
2350 length=LEN(main$)
```

Description

This function returns the length of the string given as the argument.

Syntax

<num-var>=LEN(<string>)

Associated keywords

`LEFT$`, `MID$`, `RIGHT$`, `INSTR`

Demonstration program

```
300 PRINT "Type in your word";
310 INPUT A$
320 Length=LEN(A$)
325 C$=""
330 FOR V=1 TO length
340 B$=MID$(A$,V,1)
350 C$=C$+CHR$(ASC(B$)+1)
360 NEXT V
370 PRINT "The coded version is ";C$
```

In the above program each letter is copied one at a time into `B$`. Then its ASCII

value is calculated, 1 is added to the ASCII value and the new ASCII value is converted back into a character which is then added onto `C$`. See the keyword `ASC` for more information about the ASCII code.

LET

Purpose

In BASIC we often write things like

`X=6`

meaning put 6 into the box labelled X in the computer. The fact that we are changing the contents of the variable X can be made clearer by writing

`LET X=6`

The statement `X=X+6` is impossible in mathematical terms. How can something be the same as itself plus 6? In BASIC though it is quite legal to say `LET X=X+6` since the instruction simply means

'store in the variable X whatever is already there plus 6', or 'increase the value of X by 6'.

The word `LET` is optional, but its use makes the program more readable.

Examples

`100 LET length=15`

`980 LET DAY$ ="Tuesday"`

`210 IF A=6 THEN LET length=12`

Description

`LET` is an optional assignment statement.

Note: `LET` may not be used during the assignment of the pseudo-variables `LOMEM`, `HIMEM`, `PAGE`, `PTR#`, `TIME`.

Syntax

[`LET`]<var>=<expression>

LIST

Purpose

This command makes the computer list whatever program it has in its memory. It is often used before typing `RUN` to ensure that there aren't any typing errors in the program just entered.

You can list a single line

`LIST 280`

or a range of lines

`LIST 100,450`

or the whole program

`LIST`

`LIST ,400` will list all lines up to and including line 400.

`LIST 400,` will list all lines beyond line 400.

If you have a very long program you may see the whole listing whiz past before you have time to read it. To stop it, and to make the computer stop at the bottom of each page you can type **CTRL N** (while holding down the key marked **CTRL** press the letter N). Then type `LIST`. This is called 'page mode' and the computer stops at the bottom of each page. The next page will be printed when the **SHIFT** key is pressed.

To return to 'scroll mode' type **CTRL O** (hold CTRL down while briefly pressing O). Pressing **CTRL** and **SHIFT** together immediately pauses a listing, and pressing **ESCAPE** will stop a listing so that corrections can be made to the program.

If you want a listing on the printer then you can turn the printer on by typing **CTRL B** before typing `LIST`.

To turn the printer off afterwards type **CTRL C**.

`LIST` is a command and cannot be used as part of a program or as part of a multiple statement line.

The layout of programs as listed can be controlled by the command `LISTO` (see next entry). As an option, the computer can be instructed to insert spaces for the duration of all `FOR...NEXT` and `REPEAT...UNTIL` loops.

Examples

`LIST`

`LIST 400`

`LIST 400,500`

`LIST ,900`

`LIST 900,`

Description

A command which lists the current program.

Syntax

`LIST`[`,`][<num-const>][`,`][<num-const>]

Associated keywords

`NEW`, `OLD`, `LISTO`

LISTO list option

Purpose

When a program is listed on the screen or the printer, it is often convenient to show all loops within the program indented. `LISTO` can be used to control the way that the `LIST` command displays a program on the screen. It can cause the computer to insert spaces in three situations:

– After the line number.
– During `FOR...NEXT` loops.
– During `REPEAT...UNTIL` loops.

The number following `LISTO` should be in the range 0 to 7.

0 Implies no inserted spaces.
1 Implies a space after the line number.
2 Implies spaces during `FOR...NEXT` loops.
4 Implies spaces during `REPEAT...UNTIL` loops.

The numbers which select each option (1, 2 or 4) can be added together to select multiple options. If spaces were required during `FOR...NEXT` and `REPEAT...UNTIL` loops then `LISTO6` would be selected. `LISTO7` puts a space after the line number and double spaces for `FOR...NEXT` and `REPEAT...UNTIL` loops.

The most common options are `LISTO0` and `LISTO7`.

When editing programs using the cursor editing keys it is strongly advised that you use the `LISTO0` option or else you will **COPY** in a lot of extra space.

Description

`LISTO` affects the print format produced by subsequent `LIST` commands. Bit 0 of the argument controls the single space after the line number; bit 1 the double space in `FOR...NEXT` loops; bit 2 the double space in `REPEAT...UNTIL` loops.

Syntax

`LISTO`<num-const>

Associated keywords

`LIST`

LN natural logarithm

Purpose

A mathematical function to calculate logarithms to the base e – usually called 'natural logarithms'.

Examples

```
100 X=LN(temp)
```

```
3000 H5=LN(REDOXpotential)
```

Description

A function returning the natural logarithm of its argument. Inverse logarithms (anti-logarithms) can be calculated by using

```
antilog = EXP(log)
```

Syntax

<num-var>=**LN**<numeric>

Associated keywords

LOG, EXP

LOAD

Purpose

To load a program into the computer from cassette tape, disc or Econet, whichever is the current filing system. For example

```
LOAD "GAME1"
```

Once the program has been loaded, type `RUN` to start it.

When you use the word `LOAD`, the computer forgets any previous program it had in memory and also the values of all variables.

If you are loading from disc then the file name (enclosed in quotes) must be a string of not more than seven characters in length (or ten characters for the Advanced Disc Filing System). If a disc directory is specified then you do this by putting the directory character before the file name, like this:

```
LOAD "B.GAME1"
```

If you wish to load from a drive other than the one currently selected then the drive number also is included in the quotes preceded by a colon. For example

```
LOAD ":0.D.GAME1"
```

will load a file called `GAME1` in directory `D` from drive `0`.

If you are loading from cassette, then the computer will show the name of each section of the program as it finds it on the cassette. The file name (enclosed in quotes) may be up to ten characters in length. `LOAD""` (with no file name) will load the next program found on cassette, whatever its name. This does not work on disc or Econet or other filing systems.

`LOAD` does not run a program. It just loads a file into memory. It clears all variables except A% to Z% and `@%`. The command `LOAD` cannot be used in a program.

The statement `CHAIN` can be used in a program (or as a command) to load another program and to start that program running automatically.

Examples

```
LOAD "STARWARS"
```

```
LOAD "MYPROG"
```

Description

The command `LOAD` deletes the current program, clears all variables except the resident integer variables and then loads a new program from the current filing system. The program to be loaded must be in internal format.

Since `LOAD` is a command it cannot form part of a multiple statement line.

Syntax

`LOAD`<string>

Associated keywords

`SAVE`, `CHAIN`

LOCAL

Purpose

This informs the computer that the named variables are 'local' to the procedure or function in which they occur; their use in this procedure or function in no way affects their value outside it. See the keyword **DEF** for more information.

Example

```
560 LOCAL X,Y,A$,B$
```

Description

A statement which can only be used inside a procedure or function definition. **LOCAL** saves the values of the external variables named and restores these original values when the function or procedure is completed.

Syntax

LOCAL<string-var>|<num-var>{,<string-var>|<num-var>}

Demonstration procedure

```
780 DEF PROCdrawTRIANGLE(size)
790 LOCAL X1,X2,Y1,Y2
800 X1=320-size
810 X2=320+size
820 Y1=256-size
830 Y2=256+size
840 MOVE X1,Y1
850 DRAW X2,Y1
860 DRAW 320,Y2
870 DRAW X1,Y1
880 ENDPROC
```

Associated keywords

DEF, ENDPROC, FN, PROC

LOG logarithm

Purpose

A mathematical function to calculate the common logarithm of a number to base 10.

Examples

```
100 Y=LOG (y)

440 pressure=LOG(speed)
```

Description

A function giving the common logarithm to base 10 of its argument. Inverse logarithms (anti-logarithms) can be calculated by using

```
antilog = 10^log
```

Syntax

<num-var>=LOG<numeric>

Associated keywords

LN, EXP

LOMEM

Purpose

Different sections of the computer's memory are used for different purposes. Normally BASIC makes an intelligent decision about where to store the numbers that the user calls X and Y etc. In fact it stores these variables immediately after the user's program. You can change the place where it starts to store these variables by changing the value of **LOMEM**, but this must be done right at the beginning of the program.

The variable **LOMEM** gives the address of the place in memory above which the computer stores all its variables (except for the resident integer variables **@%** and A% to Z%.

LOMEM is normally set to be the same as **TOP** which is the address of the top of the user program. See the keyword **HIMEM** and the memory map in Appendix J for more details.

Do not accidentally move **LOMEM** in the middle of a program – the interpreter will lose track of all the variables that you are using.

Examples

```
100 LOMEM=TOP+&100
```

```
PRINT LOMEM
```

```
PRINT ~LOMEM
```

Note: The **~** tells the computer to print the value in hexadecimal.

Description

A pseudo-variable which sets the place in memory above which the BASIC interpreter stores dynamic variables – those that are created and destroyed as required. Space is always set aside for the resident variables **@%** to Z%. Normally **LOMEM** is set equal to **TOP** which contains the address of the end of the user program.

Moving **LOMEM** in the middle of a program will cause loss of all variables.

Syntax

LOMEM=<numeric>
or

<num-var>=**LOMEM**

Associated keywords

HIMEM, TOP, PAGE

MID$

Purpose

To copy part of one string into another string. For example, if

```
demo$="DOGMATIC"
```

then the middle part of **demo$**, starting at position four and going on for three letters, ie

```
MID$(demo$,4,3)
```

would equal **MAT**. In fact **MID$** can be used to copy any part of a string – not just the middle part. Thus

```
MID$(demo$,1,3)
```

would equal **DOG** and

```
MID$(demo$,5,4)
```

would be **ATIC**.

This string function is very useful for selecting one word out of a long line. There is a demonstration program under the keyword **GOSUB** and another under the keyword **LEN**.

If the last number is omitted then the function returns with the rest of the string.

Example

```
RESTofLINES=MID$(main$,10)
```

Description

A string function which returns a sub-section of the first argument's string. The second argument gives the starting position and the third argument gives the number of characters to be copied. If the source string is too short then the function returns as many characters as possible from the starting position.

Syntax

<string-var>=**MID$**(<string>,<numeric>[,<numeric>])

Associated keywords

LEFT$**, **RIGHT$, **LEN**, **INSTR**

MOD modulus

Purpose

The function **MOD** gives the remainder after division. When doing division with whole numbers (I emphasise – with *whole* numbers) it is sometimes useful to know the remainder. For example 14 divided by 5 leaves a remainder of 4 (14=2*5+4). Similarly

PRINT 14 MOD 5

would print **4**. The whole number part of the above division is given by the function **DIV**. Thus

PRINT 14 DIV 5

would print **2**.

Notice that the result of both **DIV** and **MOD** is always a whole number.

In fact *all* numbers used in the calculation of the function are first converted to integers (using internal truncation) before the computer calculates the result.

Thus

```
14   DIV 5    gives 2
14.6 DIV 5.1  gives 2
14   MOD 5    gives 4
14.6 MOD 5.1  gives 4
```

The second example (**14.6 DIV 5.1**) is really the same as the first. However

14.6 DIV 4.9 gives **3** and

14.6 MOD 4.9 gives **2**

are quite different. In effect the computer sees them as

```
14 DIV 4
14 MOD 4
```

Examples

100 LET X=A MOD B

PRINT length MOD 12

Description

A binary operation giving the signed remainder of an integer division. `MOD` is defined such that

```
A MOD B = A-((A DIV B)*B)
```

Syntax

<num-var>=<numeric>**MOD**<numeric>

Associated keywords

`DIV`

MODE graphics mode

Purpose

This statement is used to select which display **MODE** the computer is about to use. Changing **MODE**s clears the screen.

Mode	Graphics	Colour	Text
0	640x256	Two colour display	80x32 text
1	320x256	Four colour display	40x32 text
2	160x256	16 colour display	20x32 text
3		Two colour text only	80x25 text
4	320x256	Two colour display	40x32 text
5	160x256	Four colour display	20x32 text
6		Two colour text only	40x25 text
7		Teletext display	40x25 text

MODE 7 uses the Teletext standard display characters. These cannot be changed by the user. Since these characters differ slightly from the standard ASCII set you will find that a number of characters on the screen do not correspond to those printed on the keys. For example a left hand square bracket will be displayed as an arrow.

In **MODE**s **0** to **6** the character set can be changed by the user. See **VDU23** in chapter 34.

You cannot change **MODE** inside a procedure or function.

MODEs **128** to 135 are the 'shadow' equivalents of **MODE**s **0-7**. See chapter 42 for more details.

Examples

```
10 MODE 5
```

```
MODE 7
```

Description

A statement used to select the display **MODE** ,which may not be used in a procedure or function. **MODE** resets the value of **HIMEM**, except when a second processor is in use, or when the computer is operating in the shadow screen mode.

Syntax

`MODE`<numeric>

Associated keywords

`CLS`, `CLG`, `HIMEM`

MOVE

Purpose

This statement moves the graphics cursor to a particular absolute position without drawing a line. For example to move to a point 100 points across the screen and 300 points up the screen one would say

`MOVE 100,300`

Examples

`1050 MOVE 100,300`

`MOVE X,Y`

Description

To move the graphics cursor to a new position without drawing a line. This statement is identical to `PLOT4`.

Syntax

`MOVE<numeric>,`<numeric>

Associated keywords

`DRAW`, `MODE`, `GCOL`, `PLOT`

NEW

Purpose

To 'remove' a program from the computer's memory. In fact the program is still there but the computer has been told to forget about it. If you want to, you can usually recover the old program by typing `OLD`. This only works if you have not entered any part of another program.

`NEW` is normally used as a command before typing in a new program - to ensure that the computer has forgotten all its previous instructions.

`NEW` does not clear any of the resident integer variables A% to Z% or `@%`.

Example

`NEW`

Description

A command which resets internal pointers to 'delete' all program statements. The program may be recovered with `OLD` provided no new statements have been entered and no new variables have been created. Since it is a command it cannot form part of a multiple statement line.

Syntax

`NEW`

Associated keywords

`OLD`

NEXT

Purpose

This is used in conjunction with **FOR** to make the computer loop around a set of statements a number of times.

If the loop is opened with (for example)

```
FOR speed=10 TO 100
```

then the **NEXT** statement would normally be in the form

```
NEXT speed
```

but the word **speed** is optional.

Example

```
340 length=100
350 FOR X=0 TO 640 STEP 2
360 Y=2*length+250
370 DRAW X,Y
380 NEXT
```

Description

A statement delimiting **FOR...NEXT** loops. The control variable (**X** in the last example) is optional.

If a variable is given after **NEXT** then the computer will 'pop' other **FOR...NEXT** loops off the 'stack' until it finds a matching variable. If none is found, an error will be reported.

Syntax

NEXT[<num-var>]**,**[<num-var>]**,**...

Associated keywords

FOR, **TO**, **STEP**

NOT

Purpose

This is normally used with an **IF...THEN** statement to reverse the effect of some test.

Example

680 IF NOT (A=6 AND B=5) THEN PRINT "WRONG"

If **A=6** and **B=5** then the computer will not print **WRONG**.

Description

NOT is a high priority unary operator equivalent to unary minus.

Syntax

<num-var>=**NOT**<numeric>

or

<testable condition>=**NOT**(<testable condition>)

OLD

Purpose

To recover a program which has been recently deleted by `NEW` or by pressing the **BREAK** key. Programs can only be recovered if no program lines have been entered and if no new variables have been created since the program was deleted. If you get the message `Bad program`, then type `NEW` again.

Typing `NEW` or pressing **BREAK** are quite drastic moves. `OLD` will do its best to recover your program but will not always succeed fully. In particular if the first line number in your program is greater than 255 then it will get that one line number wrong. The **ESCAPE** key provides a clean method of stopping a program. **BREAK** is much more violent and should be avoided.

Example

`OLD`

Description

A command which undoes the effect of `NEW`.

Syntax

`OLD`

Associated keywords

`NEW`

ON

Purpose

To alter the order in which BASIC executes a program by jumping to one of a selection of lines depending on the value of a particular variable. The word ON is used with three other keywords GOTO, GOSUB and ERROR. For example:

```
ON value GOTO 800,920,100 ELSE 7300
ON result GOSUB 8000,8300,120,7600
ON ERROR GOTO 9000
ON ERROR GOSUB 2001
```

First:

```
ON X GOTO 1100,1210,1450,1600,1950
```

If the value X is equal to 1 then the program will go to line 1100. If X=2 then the program will go to line 1210. If X=3 then line 1450 and so on.

What is it used for? Suppose that you are counting coins put into a machine and you want to offer different things if one, two or three coins are put in. The program which follows illustrates, in outline, how ON GOTO will help.

```
450 REM the variable COINS gives the number
460 REM of coins inserted
500 ON COINS GOTO 550,600,650
550 PRINT "One coin buys a biscuit"
560 REM gives him a biscuit somehow
590 GOTO 1000
600 PRINT "Two coins can buy tea or coffee"
610 GOTO 1000
650 PRINT "Three coins can buy a piece of cake"
660 REM something else in here as well
690 GOTO 1000
1000 REM all the routines end up here
```

Secondly:
```
ON X GOSUB 2200,2300,2400,2500
```

ON can also be used with GOSUB instead of GOTO. See GOSUB for an explanation of subroutines.

ON X GOSUB provides a neat way of using different subroutines in different situations.

An **ELSE** clause can be included at the end of **ON GOTO** and **ON GOSUB** to trap out of range values without causing an error.

Thirdly:

```
ON ERROR GOTO
ON ERROR OFF
```

If the computer detects an error in your program or in the disc drives or anything else that it can't handle, then it produces an error. In other words it complains and stops. The complaint takes the form of a message on the screen - for example **Too big**.

Sometimes it is vital that the computer looks after such situations without troubling the user. The statement **ON ERROR GOTO 7000** ensures that if an error occurs the computer does not complain and does not stop. Instead it goes to a section of program at line 7000 (in this case) which has been specially written to get the computer out of the mess it is in. This section of program may have to give the user instructions like **Please enter a smaller number** or it may be able to sort out the problem in some other way.

How well this 'error trapping' works depends on the skill of the programmer in thinking of every possible thing that can go wrong. You will soon re-discover Murphy's Law:

'If anything can go wrong, it will.'

Good error handling is vital in all programs for use by non-specialists - and that means most people!

The statement **ON ERROR OFF** lets the computer deal with errors once again - cancelling the effect of **ON ERROR GOTO**.

Examples

```
40 ON ERROR GOTO 9000

50 ON ERROR PRINT "The computer is confused"

10 ON ERROR GOSUB 2000
```

Description

A statement providing multiple options in changing the order of execution of a program, and error trapping.

Syntax

ON<num-var>**GOTO**<numeric>**{,**<numeric>**}**[**ELSE**<statement>]

or

ON<num-var>**GOSUB**<numeric>**{,**<numeric>**}**[**ELSE**<statement>]

or

ON ERROR<statement>

or

ON ERROR OFF

Associated keywords

GOTO, **GOSUB**, **ELSE**

OPENIN open file for input to computer (from cassette, disc or Econet)

Purpose

To tell the computer that the program wishes to read data (words and numbers). Reading data in is quite a complicated procedure for the computer and it needs advance warning when you wish to do so. The advance warning is given by the **OPENIN** keyword.

One use of this facility is to store names and addresses on file (eg the cassette or disc) and to read the file in each time you want to use it. After you have corrected it you can then transfer it back to disc (using **OPENOUT**) where it will be saved for future use. Further information about cassette, disc and Econet files is provided in chapter 31.

A typical example of the use of **OPENIN** is

```
X=OPENIN("cinemas")
```

This informs the computer that you will shortly want to read data in from a file which is recorded under the name 'cinemas'. The file name is 'cinemas'.

In accepting this instruction the computer allocates a 'channel' to this operation. It is as if it said 'OK that information will be provided on telephone number 6'. It makes X=6 (or whatever number it decides). In all future operations on that file you must refer to it as channel X (channel 6 in this example).

You get the actual data into the computer by using either **BGET#X** or **INPUT#X** as the demonstration program on the next page indicates.

Example

```
230 file=OPENIN("census")
```

Description

A function which attempts to open a file for input or random access. In a disc or Econet environment if a file already exists with the correct name it will be opened for reading.

The function returns the channel number allocated by the computer's filing system. If the file does not exist then zero is returned.

Syntax

<num-var>=**OPENIN**(<string>)

Associated keywords

OPENOUT, OPENUP, EXT#, PTR#, INPUT#, PRINT#, BGET#, BPUT#, EOF#, CLOSE#

Demonstration program

```
 10 REM to read in the names of 10 cinemas from
 20 REM disc assuming of course that you put
 30 REM them there sometime before!
 50 REM dimension a string array of 10 slots
 60 DIM cine$(10)
 90 REM open the file
100 channel=OPENIN ("CINEMA")
110 REM and read in the ten cinema names
120 FOR X=1 TO 10
130 INPUT# channel, cine$(X)
140 NEXT X
150 REM that's the information in
160 REM do whatever you want with it!
```

OPENOUT open file for output to cassette, disc or Econet

Purpose

This opens a cassette or disc file for output. Before you can record data (rather than programs) you have to open a file. More information about files is given in chapter 31.

OPENOUT is used to inform the computer that you wish to record data on cassette or disc. The computer allocates a channel to the operation.

When working with discs or over Econet then if a file already exists with that name it will be deleted. If no file exists then a new one will be created.

Example

```
330 X=OPENOUT("cinemas")
```

Description

A function which returns the channel number allocated to an output file.

If a file of the same name exists then that file will first be deleted. If no file exists then one will be created.

Syntax

<num-var>=**OPENOUT**(<string>)

Associated keywords

OPENIN, **OPENUP**, **PTR#**, **EXT#**, **INPUT#**, **PRINT#**, **BGET#**, **BPUT#**, **EOF#**, **CLOSE#**

OPENUP open a file for update

Purpose

This statement can be used with disc or Econet systems to open a file for update – that is, simultaneous reading and/or writing. With Econet, only one user may open a file for writing at any one time and therefore **OPENUP** should only be used in single user environments.

If a file of the given name exists already then that file will be opened without any changes taking place to the file. If no file of that name exists then **OPENUP** will fail to open the file requested.

OPENUP is normally used with random access files on disc or on the Level 2 Econet filing systems.

Example

```
500 Y% = OPENUP("DATA")
```

Description

A function which returns the channel number allocated to a file opened for both reading and writing. The file must exist before this function can be used.

Syntax

<num-var>=**OPENUP**(<string>)

Associated keywords

OPENIN, OPENOUT, PTR#, EXT#, INPUT#, PRINT#, BGET#, BPUT#, EOF#, CLOSE#

OPT option

Purpose

This statement determines what output is produced on the screen when assembly language routines are processed by the BASIC interpreter. An understanding of the operation of assemblers is required to understand the following.

During assembly two common errors can occur: **Branch out of range** and **Unknown label**.

The latter will occur during pass one for all forward references. It is therefore often desirable to turn off assembler error messages during pass one.

The statement **OPT** is followed by a number in the range 0 to 7, with the following results:

0 Assembler errors suppressed, no listing.
1 Assembler errors suppressed, listing.
2 Assembler errors reported, no listing.
3 Assembler errors reported, listing.

Options 4, 5, 6 and 7 behave exactly as options 0, 1, 2 and 3 except that the code can be placed at a different location from that at which it is intended to execute. With options 4 to 7 the variable P% controls the program counter during assembly, and the variable O% gives the memory location where the code is placed.

The **OPT** statement can only occur inside the square brackets which enclose Assembly Language commands. **OPT** is set to 3 every time the BASIC interpreter finds a **[**. Do not confuse it with ***OPT** which is described in chapter 43.

Examples

```
200 OPT 1

350 OPT (pass*2+list)
```

Description

An assembler pseudo-operation controlling the output during assembly. **OPT** is followed by an expression as detailed above.

Syntax

OPT<numeric>

Demonstration program

```
10 oswrch=&FFEE
20 DIM memory% 100
30 FOR Z=0 TO 3 STEP 3
35 P%=memory%
40 [OPTZ
50 .start LDA#ASC"!"
60 LDX #40
70 .loop jsr oswrch
80 dex:BNE loop
90 rts:] NEXT Z
100 CALL start
110 END
```

OR

Purpose

To enable one condition *or* another condition to determine what happens next. The **OR** operator can be used either as a 'logical OR' or as a 'Boolean OR'. See the keyword **AND** for details of logical and Boolean operators.

Example

```
75 IF X=6 OR date>20 THEN PRINT "Good"
```

Description

An operator performing Boolean integer logical **OR** between two numerics.

Syntax

<num-var>=<numeric>**OR**<numeric>

Associated keywords

AND, EOR, NOT

OSCLI operating system command line interpreter

Purpose

It is very useful in a BASIC program to be able to send commands to the operating system. Such commands might include ***FX** commands followed by two numbers. When the program is written you do not always know which numbers are to follow the ***FX** statement. However, you cannot substitute variables such as X and Y directly after the ***FX** because these variables are not known to the command line interpreter but are only known to the BASIC language. Thus the statement

```
X=5:Y=3:*FX X,Y
```

would be meaningless to the operating system. The statement **OSCLI** provides a neat way of passing variables to the operating system in such cases. **OSCLI** is followed by a string variable which is set to contain the values to be passed to the operating system. Note that numbers must be converted to string form by using the **STR$** function; the above example would work correctly with the following.

```
10 X=5
20 Y=3
30 A$="FX "+STR$X+","+STR$Y
40 OSCLI A$
```

Examples

```
10FN$="XYZ" : REM FILE NAME
20 START%= &4000 : REM START OF CODE
30 FINISH%= &6000 : REM END OF CODE
40 EXECADD%= &5000 : REM EXECUTION ADDRESS
300 OSCLI "SAVE "+FN$+" "+STR$~(START%)+"
"+STR$~(FINISH%)+" "+STR$~(EXECADD%)
```

Note that no ***** is needed in the string.

Description

A statement which passes its string argument to the operating system command line interpreter.

Syntax

`OSCLI`<string>

Associated keywords

`STR$`, `CHR$`

PAGE

Purpose

`PAGE` is a variable which gives the address in memory where BASIC has stored (or will store) the user's program. This is usually automatically set to be the lowest available address in the computer's Random Access Memory but can be changed by the user.

`PAGE` can be used to enable the computer to store two different programs at the same time in different areas of memory. Use with care.

Examples

```
PRINT PAGE

10 PAGE=&5000

20 PRINT ~PAGE

235 PAGE=TOP+1000
```

Description

A pseudo-variable giving the address used by the interpreter for the start of the user program. The least significant byte of `PAGE` is always set to zero by the computer. In other words user programs always start on a 'page' boundary where one page is 100 bytes hex (256 bytes decimal).

Syntax

`PAGE=<numeric>`

or

`<num-var>=PAGE`

Associated keywords

`TOP`, `LOMEM`, `HIMEM`

PI

Purpose

PI has the value 3.14159265. It is used in the example to calculate the area of a circle radius R.

Examples

```
100 AREA=PI*R^2
```

```
PRINT PI
```

Description

```
PI=3.14159265
```

Syntax

<num-var>=PI

PLOT

Purpose

PLOT is the multi-purpose point, line and triangle drawing statement in BASIC.

The first number which follows the keyword **PLOT** tells the computer what kind of point, line or triangle it is going to draw. The two following numbers give the X and Y coordinates to be used in plotting the point or drawing the line or triangle.

PLOT K,X,Y plots to the point at X,Y in a manner determined by the value of K. The effect of each value of K will be:

0   Move relative to last point.

1   Draw line relative in the current graphics foreground colour.

2   Draw line relative in the logical inverse colour.

3   Draw line relative in the current graphics background colour.

4   Move to absolute position.

5   Draw line absolute in the current graphics foreground colour.

6   Draw line absolute in the logical inverse colour.

7   Draw line absolute in the current graphics background colour.

Higher values of K have other effects which are related to the effects given by the values 0 to 7.

  8- 15 As 0-7 but with the last point in the line omitted.

16- 23 As 0-7 but with a dotted line.

24- 31 As 0-7 but with a dotted line and without the last point on the line.

32- 63 Reserved.

64- 71 As 0-7 but only a single point is plotted.

72- 79 As 0-7 but to draw a horizontal line to the left and right of the point until a colour other than the current background colour is reached.

80- 87 As 0-7 but plot and fill a triangle.

When filling solid triangles with colour the computer fills the triangle between the coordinates given and the last *two* points visited.

88- 95   As 0-7 but to draw a horizontal line to the right until reaching the current background colour.

96- 255   Reserved for future expansions.

See chapter 34 on VDU drivers for an alternative interpretation of the numbers given above.

Suppose that in the above example, **PLOT K,X,Y**, the value of X was 50 and the value of Y was 80 then 'draw line relative' would mean draw a line to the point on the screen 50 places to the right of the origin and 80 places up from the origin.

'Logical inverse colour' is explained next.

In two colour **MODE**s the logical inverse colour of logical colour 0 is logical colour 1.

In four colour **MODE**s the following apply:

Logical colour	Inverse
0	3
1	2
2	1
3	0

In the 16 colour **MODE** logical colour 0 becomes 15, logical colour 1 becomes 14 and so on.

When drawing lines the computer draws a line from the last point X,Y position given.

Normally the origin is set at the bottom left of the screen, but its position may be moved to any point by using the **VDU29** statement. See chapter 34 for more information.

The graphics screen is 1280 points (0-1279) wide and 1024 (0-1023) points high.

The most commonly used **PLOT** statements are **PLOT 4** and **PLOT 5**, so these two have been given duplicate keywords; **MOVE** and **DRAW**.

To print a string at a specific place on the screen use the **TAB(X,Y)** statement. As an alternative one can join the graphics and text cursor together with the statement **VDU5** so that the computer prints text at the graphics cursor position. Once that has been done then the graphics cursor can be moved with **MOVE**, **DRAW** and **PLOT** statements.

Examples

```
100 PLOT 3,X,Y
```

```
PLOT 6,100,220
```

Description

A statement controlling the generation of points, lines and triangles on the screen.

Syntax

`PLOT`<numeric>,<numeric>,<numeric>

Associated keywords

MODE, CLG, MOVE, DRAW, POINT, VDU, GCOL

POINT

Purpose

To find out the colour of a certain position on the screen. Suppose that you are playing a game involving moving a car around a race track. On the race track are pools of green oil. To find out if the place where your car is about to move to has oil on it (so that the car will skid) you need to be able to find out if the screen is coloured green at that point.

The number returned is the logical colour of the screen at the graphics point specified. If the selected point is off the screen then the number returned will be -1. There must not be a space between the word `POINT` and the opening parenthesis.

Examples

```
1340 colour=POINT(X,Y)
```

```
100 IF POINT (X,Y)=2 THEN PRINT "SKID!!"
```

Description

A function returning a number representing the colour on the screen at the specified coordinates. If the point is off the screen then the function returns -1.

Syntax

<num-var>=`POINT`(<numeric>,<numeric>)

Associated keywords

`PLOT`, `DRAW`, `MOVE`, `GCOL`

POS position

Purpose

This function finds out how far across the screen the flashing cursor is. The left hand side of the screen is position 0 and the right hand side is position 19, 39 or 79 depending on the **MODE** that has been selected.

Examples

```
1005 X=POS
```

```
320 distance=POS
```

Description

A function returning the horizontal position of the cursor in the current text window.

Syntax

<num-var>=POS

Associated keywords

COUNT, **TAB**, **VPOS**

Demonstration program

To print spaces on the screen up to a certain horizontal position – for example to align columns.

```
100 column=12
110 REPEAT PRINT"";
120 UNTIL POS=column
```

PRINT

Purpose

This does not print anything on paper. It does, however, print words and numbers on the screen.

Anything enclosed in inverted commas (**" "**) will be printed exactly as it is.

Things not enclosed in inverted commas will be assumed to be variable names and the contents of the variable will be printed out. The exact layout of the numbers and figures on the screen will depend on the punctuation used in the **PRINT** statement.

The items following the word **PRINT** are referred to as the 'print list'.

The screen display is divided into vertical strips (or fields) which are (initially) ten characters wide.

A comma after an item in the print list will cause enough spaces to be printed to ensure that the next item will be printed in the next field.

A semi-colon after an item in the print list will cause the next item to be printed on the same line and immediately following the previous item.

If the print list does not end with a semi-colon then the next **PRINT** statement will print its output on a new line.

PRINT by itself leaves a blank line. A new line can be forced at any stage in the print list by inserting an apostrophe.

The table below gives examples as they would appear, except that commas have been inserted where spaces would be to aid counting.

Example

	Print position
	12345678901234567890

```
PRINT 1,2                            1         2
PRINT 10,200                        10       200
PRINT;10;200            10200
PRINT
PRINT "Answer";A        Answer42
PRINT "Answer"A         Answer           42
PRINT "Answer",A        Answer                  42
PRINT 1/2                      0.5
PRINT 1/3               0.333333333
PRINT 3.3'2.25                3.3
                             2.5
```

The printer can be turned on at any time by typing **CTRL B** or by the statement **VDU2** in a program. The output of all **PRINT** statements will then appear on the printer as well as the screen. **CTRL C** turns the printer output off. See chapter 38 for more information about the printer.

Considerable flexibility has been built into the interpreter to enable it to print numbers in several different layouts. There is no need to learn to use these options at first but they will be invaluable when layout is crucial. A more detailed explanation of the advanced features is given below.

It is possible to control the overall field width, the total number of figures printed and the number of decimal places printed.

All these features are set with one variable called **@%**. In brief, setting

@%=131594 will give two decimal places

@%=131850 will give three decimal places

@%=&90A will return to the normal output format.

For a detailed understanding of the format it is best to consider **@%** as a four byte number (eg **@=&01020903**), each byte controlling one aspect of the print format. The most significant byte will be called B4. It has a value of 01 in the example above. The least significant byte is called B1 and has the value 03 in the example above.

B4 is tested by the function **STR$** to determine the format of strings created by that function. If B4=01 then strings will be formatted paying attention to the setting of **@%** otherwise **@%** will be ignored by **STR$** . Initially B4=00.

B3 selects the basic format thus:

00 General format (G format)

01 Exponent format (E format)

02 Fixed format (F format)

In G format numbers that are integers will be printed as integers. Numbers in the range 0.1 to 1 will be printed as 0.1 etc. Numbers less than 0.1 will be printed in exponent format.

Exponent format will always print numbers in scientific notation; 100 becomes 1E2, 1000 becomes 1E3 and 1200 becomes 1.2E3

Fixed format prints numbers with a fixed number of decimal places. If the number cannot be fitted into the selected field width it reverts to G format. The decimal points are aligned vertically which is ideal for scientific and accounting programs.

B2 controls the total number of digits printed in the selected format. If B2 is too large or too small for the **MODE** selected then B2 is taken as 10. The number is rounded to fit in the B2 digit field.

In G format B2 gives the maximum number of digits that can be printed before reverting to E format. Range 1-9.

In E format B2 specifies the total number of digits to be printed before and after the decimal point – but not counting the digits after the E. Another way of looking at it is to say that (B2-1) digits will follow the decimal point. In E format three characters or spaces always follow the final E. Range of B2 in E format is 1-10.

In F format B2 specifies the number of digits to follow the decimal point. Range 0-10.

B1 sets the overall print field width and may have any value in the range 0 to 255 which in hexadecimal is &00 to &FF.

For example accounting purposes would often require fixed format two decimal places and ten character field width.

The four bytes of **@** are built up thus:

```
@%=&  00 00 00 00
```
B4 - zero **00**
B3 - fixed format **02**
B2 - two decimal places **02**
B1 - character field **0A**

So **@%=&0002020A**, the **&** indicating that the number is in hexadecimal. You can, of course, omit the leading zeros.

Here are some other formats:

Format	(G2)	(G9)	(F2)	(E2)
@%=&	0000020A	0000090A	0002020A	0001020A
100	1E2	100	100.00	1.0E2
10	10	10	10.00	1.0E1
1	1	1	1.00	1.0E0
0.1	0.1	0.1	0.10	1.0E-1
0.01	1E-2	1E-2	0.01	1.0E-2
0.0005	5E-3	5E-3	0.01	5.0E-3
0.001	1E-3	1E-3	0.00	1.0E-3
0	0	0	0.00	0.0E0
-10	-10	-10	-10.00	-1.0E1

Description

A statement causing numeric and string values to be printed on the screen.

Syntax

PRINT{['][,?;]<string>?<numeric>}['][;]

Associated keywords

PRINT#, **TAB**, **POS**, **STR#**, **WIDTH**, **INPUT**, **VDU**

PRINT#

Purpose

This records numbers and words on cassette or disc. In other words it stores data on a file. Numbers and strings are stored in a special internal format. Before this statement is used the file must have been opened using the **OPENIN**, **OPENOUT** or **OPENUP** statements. See chapter 31 on file handling for more information.

Example

```
PRINT# file, X,Y,Z,A$,"Monday",33
```

Description

A statement which writes data to files. All values are written in a special internal format:

Integer variables are written as &40 followed by the twos complement representation of the integer in four bytes, most significant byte first.

Real variables are written as &FF followed by four bytes of mantissa and one byte exponent. The mantissa is sent lowest significant bit (LSB) first. 31 bits represent the magnitude of the mantissa and 1 bit the sign. The exponent byte is in twos complement excess 128 form.

String variables are written as &00 followed by a 1 byte 'byte count' followed by the characters in the string in reverse order.

Syntax

```
PRINT#<num-var>{,<numeric>?<string>}
```

PROC procedure

Purpose

This is used as the first part of a name to indicate that it refers to a procedure. See the keyword **DEF** for a fuller description.

Example

```
10 DEF PROChello(X)
20 LOCAL Z
30 FOR Z=0 TO X
40 PRINT "Hello - how about this for BASIC!"
50 NEXT Z
60 ENDPROC
```

Description

A reserved word used at the start of all user declared procedures. There must not be a space between **PROC** and the rest of the procedure name.

Syntax

DEF PROC<variable-name>[(<string-var>|<num-var>{,<string-var>|<num-var>}]

PROC<variable-name>[(<string-var>|<num-var>{,<string-var>|<num-var>}]

Associated keywords

DEF, **ENDPROC**, **LOCAL**

Demonstration program

```
10 REM Tower of Hanoi problem
20 INPUT "Number of disks",F
30 PROChanoi(F,1,2,3)
40 END
50 DEF PROChanoi(A,B,C,D) IF A=0 ENDPROC
60 PROChanoi(A-1,B,D,C)
70 PRINT "Move disk ";A; " from pile ";B; " to pile
" ;C
80 PROChanoi (A-1,D,C,B)
90 ENDPROC
```

PTR# pointer

Purpose

This statement is not available on cassette based systems. It selects which item in a long file is to be read or written next. Strings and numbers are stored in a long line one after the other. Each integer number occupies five bytes, each real number occupies six bytes and each string takes up the number of letters in the string plus two. See the keyword **PRINT#** for more details of the file format. The file pointer can be moved up and down the file to point to any selected word or number. Note that you have to keep a careful track of where each word or number starts to use the function. The number immediately following the keyword **PTR#** is the channel number allocated to the file when it was opened. A file must be opened with the **OPENIN** and **OPENOUT** statements. See chapter 31 for more information on file handling.

Examples

```
PRINT PTR#X

560 PTR#file=PTR#file+80

85 PTR#channel=0
```

Description

A statement and function which allows the programmer to move a pointer to a serial file and thus enables random access.

Syntax

<num-var>=**PTR#**<num-var>

or

PTR#<num-var>=<numeric>

Associated keywords

INPUT#, **PRINT#**, **BGET#**, **BPUT#**, **OPENIN**,

OPENUP, **OPENOUT**, **EXT#**, **EOF#**

RAD radian

Purpose

To convert an angle measured in degrees to radians. A radian equals approximately 57 degrees.

Examples

```
1030 X=RAD(Y)
```

```
PRINT RAD(45)
```

Description

A function converting an angular argument given in degrees to radian measure.

Syntax

<num-var>=**RAD**<numeric>

Associated keywords

```
DEG
```

READ

Purpose

To enable numbers or words that are required in a program to be made available every time the program is run. It does this by reading numbers or words into numeric or string variables from **DATA** statements in the program. Most often the data is read into an array. See the keyword **DIM** for more information on arrays. See the keyword **DATA** for a more detailed description.

Example

```
100 READ name$(X),A
```

Description

A statement which copies the next item from a data list into the variable or variables which follow the keyword **READ**. The **DATA** must contain the correct sequence of string and numeric data for the string and numeric variables to be assigned. In other words numeric data must be supplied if a numeric variable is to be filled.

Syntax

READ<num-var>|<string-var>{,<num-var>|<string-var>}

Associated keywords

DATA, **RESTORE**

Demonstration Program

```
200 INPUT"How much can you spend",AFFORD
210 PRINT"You can afford the following cars"
220 FOR X=1 TO 10:READ NAME$,PRICE
230 IF PRICE<=AFFORD THEN PRINT NAME$
240 NEXT:END
500 REM British Leyland Cars
510 DATA AUSTIN METRO 1.0 HLE, 4699
520 DATA etc etc
```

REM remark

Purpose

To enable the program writer to put remarks and comments into the program to help remember what the various parts of the program do. The computer completely ignores anything that appears after a **REM**.

When you first start writing small programs you can get away with having no **REM**s, but as your programs grow in complexity you will find it necessary to have them liberally sprinkled over your program. If you come back to a program six months after you wrote it and find no **REM**s you will have a real job trying to remember how it worked and why you used that variable name etc. Use lots of **REM**s – it will save you hours of time in the long run.

Examples

```
10 REM this revision dated 2-8-84

100 REM

550 REM data for British Leyland  cars
```

Description

This statement allows comments to be inserted in a program.

Syntax

REM<anything>

RENUMBER

Purpose

When you type in a program you give each instruction a line number. As the program develops you quite often have to insert extra lines between other lines. You might well need to insert 25 lines between line numbers 300 and 310 – difficult!

The `RENUMBER` command will go through your program and renumber it automatically. It recalculates things like `GOTO 220` – which might well become `GOTO 180` etc. However if your program contains the statement `GOTO 100` and there is no line 100 then the `RENUMBER` command will be unable to deal with the problem and will say

`Failed at line <new line number>`

If you renumber a program containing an `ON GOTO` statement which contains a *calculated* line number, eg

`ON X GOTO 120,240,2*R,1000,2000`

references prior to the calculated line will be renumbered. However it will not recalculate a calculated line number or other line numbers in the same statement–ie `2*R,100` and `2000` in the example given.

The command `RENUMBER` will renumber your program giving the first line the number 10, the second 20 and so on.

The command `RENUMBER 200` will give the first line of your program the number 200, the second will become line 210 etc.

The command `RENUMBER 200,4` would renumber starting with line 200 and then using 204, 208 etc.

`RENUMBER` is a command: it cannot be used in a program, or as part of a multiple statement line.

Examples

`RENUMBER`

`RENUMBER 100,20`

`RENUMBER 6000`

Description

`RENUMBER` is a command which renumbers a user's program and will correct most of the cross-references within the program.

Syntax

`RENUMBER`[<num-const>[`,`<num-const>]]

REPEAT

Purpose

To make the computer repeat a set of instructions a number of times until some condition is met.

If you jump out of a `REPEAT...UNTIL` loop with a `GOTO` statement (which is bad practice) you must jump back in.

A single `REPEAT` may have more than one `UNTIL`.

Example

```
10 REM print stars for 1 second
20 NOW=TIME
30 REPEAT PRINT
40 UNTIL TIME=NOW+100
```

Description

A statement which is the start of a `REPEAT...UNTIL` loop. These loops always execute once and may be nested up to a depth of 20.

Syntax

`REPEAT`

Associated keywords

`UNTIL`

REPORT

Purpose

To get the computer to report in words what the last error was.

Example

`100 REPORT`

Description

`REPORT` prints the error message appropriate to the last error condition.

Syntax

`REPORT`

Associated keywords

`ERR, ERL, ON ERROR`

RESTORE

Purpose

Sometimes it is useful to have several sets of data in one program. For example one might want information on British Leyland cars and on Lotus cars as in the example given in chapter 22. The **RESTORE** statement enables the data pointer to be moved from one set of data to the other.

The word **RESTORE** by itself resets the data pointer to the first set of data in the program.

Examples

230 RESTORE

100 RESTORE 6500

RESTORE apointer

Description

This statement can be used at any time to reset the data pointer to any selected line number.

Syntax

RESTORE[<numeric>]

Associated keywords

READ, **DATA**

RETURN

Purpose

The word **RETURN** – not the key marked **RETURN** – is used in a program a the end of a subroutine to make the computer return to the place in the program which originally 'called' the subroutine. See **GOSUB** for more details.

There may be more than one **RETURN** statement in a subroutine – but preferably there should be one entry point and one (**RETURN**) exit point.

You should try very hard to avoid leaving a subroutine with **GOTO** – you should always exit with **RETURN**. Why? Well you will soon discover in reasonable sized programs that you can get into an awful tangle and lose track of how a program works if you make the program jump all over the place.

The importance of dividing your programs into clearly defined sections wherever possible, with one entry point and one exit point, cannot be over emphasised.

Examples

```
200 RETURN
```

```
300 IF X>4 THEN RETURN
```

Description

A statement which causes the program to branch to the statement after the one which contained the **GOSUB** which called the current subroutine.

Syntax

```
RETURN
```

Associated keywords

```
GOSUB, ON GOSUB
```

RIGHT$

Purpose

To copy the right hand part of one string into another string. For example if

`ABCDE$="HOW ARE YOU"` then

`RIGHT$(ABCDE$,3)` would be `"YOU"` and

`RIGHT$(ABCDE$,7)` would be `"ARE YOU"`

Note that `RIGHT$(ABCDE$,100)` would be `"HOW ARE YOU"` since there are only 11 characters in `HOW ARE YOU`.

Examples

`A$=RIGHT$(B$,5)`

`last$=RIGHT$(last$,X)`

Description

A string function returning a specified number of characters from the right hand end of another string.

Syntax

<string-var>=`RIGHT$`(<string>`,`<numeric>)

Associated keywords

`LEFT$`, `MID$`

RND random

Purpose

To generate a random number.

What exactly this function does is determined by the number which follows the word **RND**.

RND by itself generates a random whole number between -2147483648 and 2147483647.

RND(-X) returns the value -X and resets the random number generator to a number based on X.

RND(0) repeats the last random number given by **RND(1)**.

RND(1) generates a random number between 0 and 0.999999.

RND(X) generates a random whole number between (and possibly including) 1 and X.

The parentheses are compulsory and must immediately follow the word **RND** with no intervening space.

Examples

```
PRINT RND(6)

340 largenumber%=RND

950 PRINT RND(1)
```

Description

A function generating a random number. The range of the number generated depends on the argument (if any).

Syntax

<num-var>=**RND**[(<numeric>)]

Associated keywords

None

RUN

Purpose

To make the computer obey the statements in the program in its memory.

All variables (except the resident integer numeric variables `@%` and A% to Z%) are first deleted and then the program is executed.

RUN is a statement and programs may therefore execute themselves.

If you want to start a program without clearing all the variables then you can use the statement

```
GOTO 100
```

or **GOTO** whatever line number you wish to start from, instead of **RUN**.

Examples

```
RUN
```

```
9000 RUN
```

Description

RUN is a statement causing the computer to execute the current program.

Syntax

```
RUN
```

Associated keywords

NEW, **OLD**, **LIST**, **CHAIN**

SAVE

Purpose

To save a program that is in the computer's memory onto cassette or disc. The program must be given a name – usually called its file name. The file name can have up to seven letters and numbers for the Disc Filing System, or ten letters and numbers for the Cassette Filing System and the Advanced Disc Filing System. The name must start with a letter, and cannot contain spaces or punctuation marks.

Examples

```
SAVE "FRED"
```

```
SAVE A$
```

Description

A command which saves the current program area – that is the area between the address given in the variables `PAGE` and `TOP.`

Syntax

`SAVE`<string>

Associated keywords

`LOAD`, `CHAIN`

SGN

Purpose

This determines whether a number is positive, zero or negative. The function returns

-1 for negative number
0   for zero
+1 for positive number

Examples

```
100 X=SGN(Y)
```

```
230 result=SGN(difference)
```

Description

A function returning -1 for an argument which is negative, +1 for a positive argument and zero for an argument equal to zero.

Syntax

<num-var>=SGN(<numeric>)

Associated keywords

ABS

SIN sine

Purpose

This calculates the sine of an angle. The angle must be expressed in radians rather than degrees – but you can convert from degrees to radians using the function **RAD**.

Examples

```
120 Y=SIN(RAD(45))

2340 value=SIN (1.56)
```

Description

A function giving the sine of its argument. The argument must be in radians.

Syntax

<num-var>=**SIN**(<numeric>)

Associated keywords

COS, **TAN**, **ACS**, **ASN**, **ATN**, **DEG**, **RAD**

Demonstration program

To draw a sine wave on the screen.

```
10 MODE 4
20 FOR X=0 TO 1280 STEP 4
30 DRAW X,500+500*SIN(X/50)
40 NEXT X
```

SOUND

Purpose

This statement is used to make the computer generate sounds using the internal loudspeaker. The sound generator is capable of making four sounds at once. Each of the four sound channels can generate one note. The keyword **SOUND** must be followed by four numbers which specify:

– Which sound channel is to be used.
– The loudness of the note (or the envelope number).
– The pitch of the note.
– How long the note is to last.

For example

```
SOUND 1,-15,52,20
```

will play a note on sound channel 1, with a loudness of -15 (maximum volume). A pitch value of 52 gives middle C and a duration of 20 will make the note last for 1 second.

```
SOUND C,A,P,D
```

The channel number (C) can be 0, 1, 2 or 3. Channel 0 is a special channel that can produce various noises, whereas channels 1, 2 and 3 are used to produce single notes. Other values of C produce special effects which are explained further on.

The amplitude or loudness (A) can have any whole number value between -15 and 4. Values -15 to 0 produce notes of fixed loudness throughout the whole note. A value of -15 is the loudest, -7 is half volume and 0 produces silence. Values of 1 to 4 enable the amplitude to be controlled while the note is playing. When you play a note on the piano the sound gradually fades away. Effects like this are selected by using one of the four user defined envelopes which are selected by setting A to be 1, 2, 3 or 4. Envelopes are explained in chapter 30 and under the keyword **ENVELOPE**.

The pitch (P) is used to set the pitch or frequency of the note. The pitch can have any value between 0 and 255. The note A above middle C is selected with a value of 88. The table in chapter 30 shows the value of P needed to produce a particular note. You will see that to go up an octave P is increased by 48 and to go up a perfect 5th P must be increased by 28.

Increasing the value of P by one will increase the note produced by a quarter of a semi-tone.

To play the chord of C major which consists of the notes C, E and G for two seconds you could enter

```
100 SOUND 1,-15,52,40
110 SOUND 2,-15,68,40
120 SOUND 3,-15,80,40
```

However to play a number of notes in succession you would enter

```
100 SOUND 1,-15,96,10
110 SOUND 1,-15,104,10
120 SOUND 1,-15,88,10
130 SOUND 1,-15,40,10
140 SOUND 1,-15,68,20
```

which plays a well-known film theme.

The duration (D) can have any value between -1 and 254. Values in the range 0 to 254 give a note duration of that number of twentieths of a second. Thus if D=40 the note will last for two seconds. Setting D=-1 means that the note will continue to sound until you actually take steps to stop it. You can either press the **ESCAPE** key or stop it by sending another note, to the same channel, which has 'flush control' set to 1 – see later in this section.

As was mentioned earlier, channel number 0 produces 'noises' rather than notes and the value of P in the statement

```
SOUND 0,A,P,D
```

has a different effect from that described for channels 1, 2 and 3. Here is a summary of the effects of different values of P on the noise channel:

P Effect

0   High frequency periodic noise.
1   Medium frequency periodic noise.
2   Low frequency periodic noise.
3   Periodic noise of frequency determined by the pitch setting of channel 1.
4   High frequency 'white' noise.
5   Medium frequency 'white' noise.
6   Low frequency 'white' noise.
7   Noise of frequency determined (continuously) by the pitch setting of channel 1.

Values of P between 0 and 3 produce a rasping, harsh note. With P set to 4 the noise is like that produced by a radio when it is not tuned to a station – a sort of 'shssh' effect. P=6 sounds like the interference found on bad telephone calls. When P is set to 3 or 7 then the frequency of the noise is controlled by the pitch setting of sound channel number 1. If the pitch of channel 1 is changed while

channel 0 is generating noise then the pitch of the noise will also change. The program below generates a noise on channel 0 and varies the pitch of the noise by changing the pitch of channel 1. Notice that the amplitude of channel 1 is very low (-1) so you will hardly hear it – but you will hear the noise on channel 0.

```
100 SOUND 0,-15,7,150
110 FOR P= 100 TO 250
120 SOUND 1,-1,P,1
130 NEXT P
```

Notice that we have not yet described how sounds can be affected by a superimposed envelope. An envelope can affect both the pitch and amplitude of a note as it is playing. Thus the statement

```
SOUND 1,-15,255,255
```

merely plays a continuous loud note, but

```
ENVELOPE 1,1,-26,-36,-45,255,255,255,127,0,0,-
127,126,0
SOUND 1,1,255,255
```

produces a complex sound controlled largely by the envelope

See the keyword **ENVELOPE** for more details.

As mentioned briefly at the start of the description of the **SOUND** statement, the channel number, C, can be given values other than 0,1,2 and 3. You do not need to understand exactly why the following works to use it!

For C you can write a four figure hexadecimal number to achieve certain effects – for example:

```
SOUND &1213,-15,52,40
```

The first parameter in the above example has the value **&1213**. The ampersand (**&**) indicates to the computer that the number is to be treated as a hexadecimal number. The four figures which follow the ampersand each control one feature. In this new expanded form the **SOUND** statement looks like

```
SOUND &HSFC,A,P,D
```

and the functions H, S, F and C will be explained in turn. In essence these numbers enable you to synchronise notes so that you can play chords effectively.

The first number (H) can have the value 0 or 1. If H=1 then instead of playing a new note on the selected channel, the previous note on that channel is allowed

to continue. If a note were gently dying away then it might be abruptly terminated when its time was up. Setting H=1 allows the note to continue instead of playing a new note. If H=1 then the note defined by the rest of the **SOUND** statement is ignored.

The second number (S) is used to synchronise the playing of a number of notes. If S=0 then the notes are played as soon as the last note on the selected channel has completed. (There is a slight simplification here; 'completed' means 'has reached the start of the release phase'.) The user is referred to the keyword **ENVELOPE** for relevant detail.

A non-zero value of S indicates to the computer that this note is not to be played until you have a corresponding note on another channel ready to be played. A value of S=1 implies that there is one other note in the group. S=2 implies two other notes (ie a total of three). If a note was sent to channel 1 with S set to 1 then it would not be played until a note was ready on another channel which also had S set to 1. For example:

```
110 SOUND &101,-15,50,200
110 SOUND 2,-15,200,100
120 SOUND &102,-15,100,200
```

When this program is run the note at line 100 will not play until channel 2 is free. Line 110 sounds a note immediately on channel 2 – and for five seconds (duration 100). When the note has completed then both the notes from lines 100 and 120 will sound together.

The third number (F) can have the value 0 or 1. If it is set to 1 then the **SOUND** statement in which it occurs flushes (throws away) any other notes waiting in the queue for a particular channel. It also stops whatever note is being generated on that channel at present. The **SOUND** statement in which F=1 then plays its note. Setting F behaves like an 'over-ride'. For example:

```
20 SOUND 2,-15,200,100
25 FOR X=1 TO 500:NEXT X
30 SOUND &12,-15,100,200
```

In the above situation line 20 will start a sound on channel 2 but this will be stopped almost immediately by line 30 which will generate a lower and longer note on channel 2. Line 25 just gives a short delay.

Setting F=1 provides an easy way of stopping an everlasting note! Thus **SOUND&13,0,0,1** stops the current note on channel 3 and instead plays one at zero loudness and of minimum length. This will stop channel 3 immediately.

The last number (C) is the channel number described earlier.

Description

The sound generator has four separately controlled synthesis channels. Each can sound at one of 16 amplitudes, including 'off'. The audio output is the sum of the channel outputs. Channels 1-3 each generate a square wave with programmable frequency. Channel 0 can produce noise (unpitched sound of pseudo-random structure) or a pulse waveform. The frequency of the pulsewave or period of the noise can be set to one of the three fixed options, or to the frequency of channel 1.

The BASIC program generates each sound by initiating one or more 'requests' each of which may take the form of a musical note or a single effect and is directed to a specific channel. If the destination channel is idle when a request requires it, the sound starts playing immediately. If a previous request is still being handled the new one is placed on a queue, where it waits until the current event is over (or past a critical stage – see **ENVELOPE**). If the queue is full, the program waits. Separate queues are provided for the four channels, each of which can hold up to four requests, not counting the one currently being executed. The program can look at the state of any queue and flush any queue, but cannot find out or alter the state of the current event, except for flushing the whole queue.

The **SOUND** keyword is followed by four parameters, the first of which consists of four hexadecimal digits. Thus

```
SOUND &HSFC,A,P,D
```

	Range	Function
H	0 or 1	Continuation
S	0 to 3	Synchronisation
F	0 or 1	Flush
C	0 to 3	Channel number
A	−15 to 4	Amplitude or envelope number
P	0 to 255	Pitch
D	1 to 255	Duration

The 'H' parameter allows the previous event on that channel to continue and if this is 1 the amplitude and pitch parameters of **SOUND** have no effect. Because the dummy note is queued in the normal way, it can be used to ensure that the release segment of sound, which occurs after the duration is over and would otherwise be truncated by the next sounding event on the same channel, is allowed to complete.

The 'S' parameter allows requests to be queued separately and then executed at the same instant, for chords and multiple voice effects. The value initially determines the number of other channels that must receive requests with the same value of S before the group will play. For example, each note of a three note chord would be generated by a `SOUND` with the value of 2 for S. The system will read the value of S from the first one and then wait for two more requests with 2 as the value of S before playing the complete chord. Single requests use 0 for S so that they play as soon as they reach the end of the channel queue.

The parameter 'F' will normally be zero, causing the request to be queued. If it is 1, the channel queue will be flushed first, so the request will sound immediately.

The parameter 'C' determines the number of the sound channel to be used.

The 'A' parameter controls the amplitude of the sound and can be used in two ways. Positive values up to 4 select the envelope (1 to 4) to be used. If the RS423 is unused then envelope numbers up to 16 may be defined and used. Zero and negative integers up to -15 directly set the amplitude of the sound, which is then fixed at this value for the duration of the note. -15 corresponds to the loudest, and 0 is 'off'.

The 'P' parameter determines the pitch of the note. It can take values from 0 to 255.

The 'D' parameter determines the total duration of sounds whose amplitude is determined explicitly by a negative or zero value of the A parameter. The duration is given in twentieths of a second. If an envelope has been selected, by a positive value of A, then the duration D determines the total of the attack, decay and sustain periods - but not of the release phase.

Syntax

`SOUND<numeric>,<numeric>,<numeric>,<numeric>`

Associated keywords

`ENVELOPE`, `ADVAL`

SPC <small>space</small>

Purpose

This statement is used to print multiple spaces on the screen. It can only be used as part of **PRINT** or **INPUT** statements. The number in parenthesis gives the number of spaces to be printed.

Examples

```
120 PRINT "Name";SPC(6);"Age"; SPC(10);"Hours"
```

```
4030 INPUT SPC(10), "Value", V
```

Description

A statement printing a number of spaces on the screen. Up to 255 spaces may be printed.

Syntax

PRINT SPC(<numeric>)
or

INPUT SPC(<numeric>)

Associated keywords

TAB, **PRINT**, **INPUT**

SQR square root

Purpose

This statement is used to calculate the square root of a number.

Examples

```
10 X=SQR(Y)
```

```
300 X=(-B+SQR(B^2-4*A*C))/(2*A)
```

Description

A function returning the square root of its argument. An attempt to calculate the square root of a negative number will produce the error message **-ve root** which is error number 21.

Syntax

<num-var>=**SQR**(<numeric>)

Associated keywords

None

STEP

Purpose

This is part of the **FOR...TO...STEP...NEXT** structure.

In the program shown below, **STEP** indicates the amount that the variable **cost** is to be increased each time around the loop. In this case the cost is to increase in steps of five units.

The step may be positive or negative.

STEP is optional. If omitted a step size of +1 is assumed – see the keyword **FOR**.

Example

```
300 FOR X=100 TO 20 STEP -2.3
```

Description

Part of the **FOR...NEXT** construct. **STEP** is optional.

Syntax

FOR<num-var>=<numeric>**TO**<numeric>[**STEP**<numeric>]

Associated keywords

FOR, **TO**, **NEXT**

Demonstration program

```
230 FOR cost=100 TO 200 STEP 5
250 production=FNtaken(cost)
260 PRINT production, cost
270 NEXT cost
```

STOP

Purpose

This statement interrupts a program which is running and prints the message

STOP at line <line number>

on the screen; otherwise the effect is identical to **END**.

STOP may occur as many times as is needed in a program.

Examples

```
2890 STOP
3080 STOP
```

Description

Causes execution of the program to cease and a message to be printed out.

Syntax

STOP

Associated keywords

END

STR$ string

Purpose

This string function converts a number into the equivalent string representation. Thus **STR$(4.6)** would give **4.6**.

STR$ is affected by the field width and format constraints imposed by the variable **@%**. The default format is format 0, field width 10 . See chapter 10.

The opposite function of converting a string into a number is performed by the functions **EVAL** and **VAL**.

Examples

20 A$=STR$(X)

5060 num$=STR$ (size)

10 PRINT STR$~(100)

Description

A string function which returns the string form of the numeric argument as it would have been printed.

Syntax

<string-var>=**STR$**(<numeric>)

Associated keywords

VAL, **PRINT**, **EVAL**, **OSCLI**

STRING$

Purpose

This produces a long string consisting of multiple copies of a shorter string.

Thus `STRING$(6,"--0")` would be `--0--0--0--0--0--0`. This function is useful for decorative features. It should be used whenever the user needs to generate a long string from identical short strings.

It is very important, to avoid wasting memory space, that strings are set to their maximum length the first time that they are allocated. This can easily be done by using `STRING$`. For example to set `A$` to contain up to 40 characters one could write

`A$=STRING$(40," ")`

`A$` can then be set back to empty using `A$=""` before use.

Examples

`400 A$=STRING$(x,pattern$)`

`560 B4$=STRING$(5,"0+")`

`PRINT STRING$(10,"hello")`

Description

A string function returning multiple concatenations of a string.

Syntax

<string-var>=**STRING$**(<numeric>,<string>)

TAB tabulation

Purpose

TAB can only be used with the keywords **PRINT** and **INPUT**. There are two versions:

TAB(X) will print spaces up to a certain column position. If the flashing cursor is beyond the required position then the cursor will move to the next line down and space across to the required column.

TAB(X,Y) will move the cursor directly to position X,Y on the screen. Note that once **TAB(X,Y)** has been used on a line **TAB(X)** may not move to the correct position on the line.

The origin (for all text commands) is at the top left of the current text area of the screen.

The left hand column of the screen is column number 0. The right hand column is column 19, 39 or 79 depending on the graphics **MODE** selected.

The top line is line number 0, the bottom line is line number 31 or 24;

If the text scrolling area of the screen is changed then the **TAB** command will still work as outlined above.

Examples

```
340 PRINT TAB(10);name$TAB(30);job$
```

```
440 PRINT TAB(20,31);value
```

```
230 INPUT TAB(10,20) "How much" cost
```

```
875 INPUT TAB(30), "Doctor's name", DOC$
```

Description

TAB with a single argument prints spaces (and a new line if necessary) to reach the specified column.

TAB with two arguments moves the cursor directly to the specified coordinates.

Syntax

`PRINT TAB`(<numeric>[`,`<numeric>])

or

`INPUT TAB`(<numeric>[`,`<numeric>])

Associated keywords

`POS`, `VPOS`, `PRINT`, `INPUT`

TAN tangent

Purpose

This mathematical function calculates the tangent of an angle.

The angle must be given in radians but may be converted to radians from degrees using the function **RAD**. A radian is about 57 degrees.

Examples

```
PRINT TAN(RAD(45))

10 Y=TAN (X)

1030 droop=TAN(load)
```

Description

A function returning the tangent of the argument. The argument must be given in radians.

Syntax

<num-var>=**TAN**<numeric>

Associated keywords

COS, **SIN**, **ACS**, **ATN**, **DEG**, **RAD**

THEN

Purpose

A keyword used with IF to decide on a course of action as the result of some test.

Examples

```
780 IF X=6 THEN PRINT "good" ELSE PRINT "bad"
```

```
200 IF A$=B$ THEN PROCgood ELSE PROCbad
```

Description

Optional part of the **IF...THEN...ELSE** structure.

Note that it is not optional if used when the condition assigns to a pseudo variable, eg

```
300 IF X THEN TIME=0
```

Syntax

IF<testable condition>[**THEN**]<statement>[**ELSE**<statement>]

Associated keywords

IF, **ELSE**

TIME

Purpose

This can be used to set or read the internal timer.

The timer counts in one hundredth of a second intervals. It is not a clock providing true time of day readout. However, once set, the internal clock will keep good time. Pressing the **BREAK** key does not reset the clock.

To convert **TIME** to a 24 hour clock use the following routines:

```
1000 SEC=(TIME DIV 100) MOD 60
1010 MIN=(TIME DIV 6000)MOD 60
1020 HR =(TIME DIV 360000)MOD 24
```

Examples

```
205 TIME=(Ho*60+Mi)*60+Se)*100
```

```
400 nowtime=TIME
```

Description

A pseudo-variable which sets or reads the lower four bytes of the internal elapsed time clock.

Syntax

TIME=<numeric>

or

<num-var>=**TIME**

Demonstration program

```
1070 finishtime=TIME+1000
1080 REPEAT
1090 REM wait for 10 seconds
1100 UNTIL TIME>=finishtime
```

TO

Purpose

Part of the **FOR...TO...STEP...NEXT** statement. The final terminating value of the loop is given after the word **TO**. See chapter 15 for further information.

Description

Part of the **FOR...NEXT** construct.

Syntax

FOR<num-var>=<numeric>**TO**<numeric>[**STEP**<numeric>]

Associated keywords

FOR, **STEP**, **NEXT**

Demonstration program

```
10 MODE 5
20 FOR C=1 TO 3
30 GCOL 3,C
40 FOR X=0 TO 1200 STEP 5*C
50 MOVE 600,750
60 DRAW X,0
70 NEXT X
80 NEXT C
```

TOP

Purpose

The function **TOP** returns the address of the first free memory location afer the user's program. The user's program is normally stored from the bottom of the available Random Access Memory upwards.

Thus the length of the user's program in bytes is given by **TOP-PAGE**.

Examples

PRINT~(TOP-PAGE):REM length in hex

2340 PRINT TOP

5460 X=TOP

Description

A function returning the first free location above the user's program.

Syntax

<num-var>=**TOP**

Associated keywords

PAGE, HIMEM, LOMEM

TRACE

Purpose

TRACE makes the computer print out the line number of each line of the program before execution.

There are three forms of **TRACE**:

TRACE ON   Causes the computer to print line numbers.

TRACE OFF   Turns off the trace facility.

TRACE 6780 Would cause the computer to report only line numbers below 6780.

With well-structured programs which have subroutines at high line numbers this will enable the user to trace through the structure of the program without being bothered with line numbers in procedures, functions and subroutines.

Note that the interpreter does not execute line numbers very often.

```
10 FOR Z=0 TO 100
20 Q=Q*Z:NEXT Z
30 END
```

would print **[10] [20] [30]** but

```
10 FOR Z=0 TO 100
20 Q=Q*Z
25 NEXT Z
30 END
```

would print **[10] [20] [25] [25] [25] [25] [25]** etc.

(Of course in **MODE 7** the **[** appears as ← and **]** appears as → .)

TRACE is also turned off after an error, or by pressing **ESCAPE** or **BREAK**.

Examples

TRACE ON

TRACE OFF

TRACE X

TRACE 3000

Description

`TRACE ON` causes the interpreter to print executed line numbers when it encounters them.

`TRACE X` sets a limit on the size of line numbers which may be printed out; only numbers less than X will be printed.

`TRACE OFF` turns trace mode off.

Syntax

`TRACE ON|OFF|`<numeric>

TRUE

Purpose

TRUE is represented by the value -1 in this computer.

Examples

`PRINT TRUE`

`300 UNTIL result = TRUE`

Description

A function returning -1.

Syntax

<num-var>=**TRUE**

Associated keywords

FALSE

UNTIL

Purpose

Part of the `REPEAT...UNTIL` construct. See the keyword `REPEAT` for more details.

Example

`450 UNTIL X<10`

Description

A program object signifying the end of a `REPEAT...UNTIL` loop.

Syntax

`UNTIL`<testable condition>

Associated keywords

`REPEAT`

USR user subroutine

Purpose

The **USR** function provides the user with a means of calling sections of machine code program which are designed to return one value. When the machine code section is called the computer sets the processor's A, X and Y registers to the least significant bytes of A%, X% and Y%. The carry flag (C) is set to the least significant bit of C%. On return from the machine code section, an integer number is generated from the four registers P, Y, X, A (most significant byte to least significant byte).

Again it must be emphasised that **USR** returns a result while **CALL** does not. Therefore you must either assign the result to a variable, eg

```
Registers=USR(&3000)
```

or print the result, eg

```
PRINT USR(&3000)
```

Each individual register may be obtained as follows:

```
10 DIM registers 3
20 !registers = USR(address)
```

After these two lines are executed,

Accumulator	=	`registers?0`
X	=	`registers?1`
Y	=	`registers?2`
Flags	=	`registers?3`

Examples

```
1400 R=USR(&3000)
```

```
670 result%=USR(plot5)
```

Description

A function allowing machine code to return directly a value for problems which do not require the flexibility of **CALL**.

Syntax

<num-var>=`USR`(<numeric>)

Associated keywords

`CALL`

VAL value

Purpose

This function takes a string which contains a number and produces the number. In other words it can convert a number represented by a string (eg **A$="+24"**) into the number.

The string must start with a plus (+) or minus (-) sign or a number. If not then the function will return zero.

The opposite function is performed by **STR$**.

Examples

450 x=VAL (length$)

1560 date=VAL(DATE$)

Description

A function which converts a character string representing a number into numeric form. If the argument is not a signed unary constant then zero will be returned.

Syntax

<num-var>=**VAL**(<string>)

Associated keywords

STR$, **EVAL**

VDU

Purpose

The statement **VDU** is followed by one or more numbers and the ASCII characters corresponding to these numbers are sent to the screen. The function **CHR$** can generate a single ASCII character from a given number. This character can be added to a string or printed. **VDU** on the other hand is used to generate a sequence of numbers that are then sent to the VDU drivers.

VDU provides an easy way of sending, for example, control characters to the VDU drivers. See chapter 34 for a detailed list of the **VDU** control codes.

Two examples will make the purpose of this statement clearer: when defining the text area of the screen four bytes have to follow the **VDU 28** statement. These four bytes represent the left X, bottom Y, right X and top Y coordinates of the text area. In **MODE 4** the range of X is 0-39 and of Y is 0-31. Thus

```
VDU 28,0,5,39,0
```

would define a six line text window at the top of the screen. If a different **MODE** is selected then the maximum screen width may be either 19, 39 or 79.

The graphics area of the screen, on the other hand, uses coordinates up to 1279 points horizontally. Thus when defining the graphics area double byte numbers must be sent to the VDU drivers since the largest number that can be sent as a single byte is 255.

```
VDU 24,0;0;1279;830
```

will define a graphics area at the bottom of the screen and 830 points high. Each of the four coordinates is sent as a double byte pair. Note that the graphics origin is bottom left while the text origin is top left and that the graphics screen is always 1280 by 1024 regardless of **MODE**.

VDU is equivalent to **PRINT CHR$;** except that it does not change the value of **COUNT**.

Examples

VDU 14 Turn auto-paging mode on.

VDU 15 Turn auto-paging mode off.

VDU 2 Turn printer on.

Description

A statement which takes a list of numeric arguments and sends them to the operating system output character routine (OSWRCH). If the argument is followed by a semi-colon then that argument will be sent as two bytes. The least significant byte will be sent first, followed by the most significant byte. This is the order required by the VDU drivers.

Syntax

`VDU <numeric>{,|;<numeric>}[;]`

Associated keywords

`CHR$`

VPOS vertical position of the cursor

Purpose

VPOS is used to find the vertical position of the text cursor on the screen.

Examples

670 V=VPOS

100 PRINT VPOS

Description

A function returning the vertical position of the text cursor.

Syntax

<num-var>=**VPOS**

Associated keywords

POS

WIDTH

Purpose

WIDTH is used to set the overall 'page width' that the computer uses. Initially this is set to zero which the interpreter interprets as 'unlimited width'.

WIDTH n will cause the interpreter to force a new line after n characters have been printed by the **PRINT** statement.

WIDTH also affects all output to the printer.

Examples

```
670 WIDTH 60
```

```
WIDTH 35
```

Description

A statement controlling the overall output field width. It is initially set to zero which disables auto new lines.

Syntax

WIDTH<numeric>

Associated keywords

COUNT

34 VDU drivers

The statement **VDU X** is equivalent to **PRINT CHR$(X);** and the statement **VDU X,Y,Z** is equivalent to **PRINT CHR$(X);CHR$(Y);CHR$(Z);**.

However the **VDU** statement is used the most when generating ASCII control codes and a detailed description of the effect of each control code is given in this chapter. The control codes are interpreted by part of the Machine Operating System called the VDU driver.

Programmers writing BASIC programs will need to refer to this summary of the VDU drivers if they want to use some of the more advanced facilities such as definition of graphics and text windows. Programmers writing other high level languages or machine code programs will also need to refer to this chapter.

The VDU drivers are part of the Machine Operating System (MOS) software. All high level languages (including BASIC) use them to print and draw on the screen. Because they are so extensive and easily accessible to programmers it will be easy to ensure that all high level languages and smaller assembly language programs have access to the same graphics facilities. There is no need for the user to write special routines to handle the screen display.

The BBC Microcomputer is designed so that it can be expanded in many ways. All expansions will be compatible with the current Machine Operating System and it is very important that those writing software use the facilities provided. In a 'twin-processor' machine the only access to the screen memory is via the 'Tube' and use of these VDU drivers and other Machine Operating System features will ensure that code will work correctly whether executed in the input/output processor or in the language processor.

The VDU drivers interpret all 32 ASCII control character codes. Many of the ASCII control codes are followed by a number of bytes. The number of bytes which follow depends on the function to be performed. The VDU code table summarises all the codes and gives the number of bytes which follow the ASCII control code.

VDU code summary

Decimal	Hex	CTRL	ASCII abbreviation	Bytes extra	Meaning
0	0	@	NUL	0	Does nothing
1	1	A	SOH	1	Send next character to printer only
2	2	B	STX	0	Enable printer
3	3	C	ETX	0	Disable printer
4	4	D	EOT	0	Write text at text cusor
5	5	E	ENQ	0	Write text at graphics cursor
6	6	F	ACK	0	Enable VDU drivers
7	7	G	BEL	0	Make a short beep
8	8	H	BS	0	Backspace cursor one character
9	9	I	HT	0	Forwardspace cursor one character
10	A	J	LF	0	Move cursor down one line
11	B	K	VT	0	Move cursor up one line
12	C	L	FF	0	Clear text area
13	D	M	CR	0	Move cusor to start of current line
14	E	N	SO	0	Page mode on
15	F	O	SI	0	Page mode off
16	10	P	DLE	0	Clear graphics area
17	11	Q	DC1	1	Define text colour
18	12	R	DC2	2	Define graphics colour
19	13	S	DC3	3	Define logical colour
20	14	T	DC4	4	Restore default logical colours
21	15	U	NAK	0	Disable VDU drivers or delete current line
22	16	V	SYN	1	Select screen mode
23	17	W	ETB	9	Reprogram display character
24	18	X	CAN	8	Define graphics character
25	19	Y	EM	5	`PLOT` k,x,y
26	1A	Z	SUB	0	Restore default windows
27	1B	[	ESC	0	Does nothing
28	1C	\	FS	4	Define text window
29	1D	]	GS	4	Define graphics origin
30	1E	^	RS	0	Home cursor to top left
31	1F	_	US	2	Move text cursor to x,y
127	7F		DEL	0	Backspace and delete

Detailed description

0 This code is ignored.

1 This code causes the next character to be sent to the printer only and not to the screen. The printer must already have been enabled with **VDU2**. Many printers use special control characters to change, for example, the size of the printed output. For example the Epson FX-80 requires a code 14 to place it into double width print mode. This could be effected with the statement

VDU1,14

or by pressing **CTRL A** and then **CTRL N**. This code also enables the 'printer ignore' character selected by ***FX6** to be sent to the printer.

2 This code turns the printer on which means that all output to the screen will also be sent to the printer. In a program the statement **VDU2** should be used, but the same effect can be obtained by typing **CTRL B**.

3 This code turns the printer off. No further output will be sent to the printer after the statement **VDU3** or after typing **CTRL C**

4 This code causes text to be written at the text cursor, ie in the normal fashion. A **MODE** change selects **VDU4**, normal operation.

5 This code causes text to be written where the graphics cursor is. The position of the text cursor is unaffected. Normally the text cursor is controlled with statements such as

PRINT TAB(5,10);

and the graphics cursor is controlled with statements like

MOVE700,450

Once the statement **VDU5** has been given only one cursor is active (the graphics cursor). This enables text characters to be placed at any position on the screen. There are a number of other effects: text characters overwrite what is already on the screen so that characters can be superimposed; text and graphics can only be written in the graphics window and the colours used for both text and graphics are the graphics colours. In addition the page no longer scrolls up when at the bottom of the page. Note however that **POS** and **VPOS** still give you the position of the text cursor. See chapter 29 for more information.

6 **VDU6** is a complementary code to **VDU21**. **VDU21** stops any further characters being printed on the screen and **VDU6** re-enables screen output. A typical use for this facility would be to prevent a password appearing on the screen as it is being typed in.

7 This code, which can be entered in a program as **VDU7** or directly from the keyboard as **CTRL G**, causes the computer to make a short 'beep'. This code is not normally passed to the printer.

8 This code (**VDU8** or **CTRL H**) moves the text cursor one space to the left. If the cursor was at the start of a line then it will be moved to the end of the previous line. It does not delete characters – unlike **VDU127**.

9 This code (**VDU9** or **CTRL I** or **TAB**) moves the cursor forward one character position.

10 This statement (**VDU10** or **CTRL J**) will move the cursor down one line. If the cursor is already on the bottom line then the whole display will normally be moved up one line.

11 This code (**VDU11** or **CTRL K**) moves the text cursor up one line. If the cursor is at the top of the screen then the whole display will move down a line.

12 This code clears the screen – or at least the text area of the screen. The screen is cleared to the text background colour which is normally black. The BASIC statement **CLS** has exactly the same effect as **VDU12**, or **CTRL L**. This code also moves the text cursor to the top left of the text window.

13 This code is produced by the **RETURN** key. However, its effect on the screen display if issued as a **VDU13** or **PRINT CHR$(13);** is to move the text cursor to the left hand edge of the current text line (but within the current text window, of course).

14 This code makes the screen display wait at the bottom of each page. It is mainly used when listing long programs to prevent the listing going past so fast that it is impossible to read. The computer will wait until a **SHIFT** key is pressed before continuing. This mode is called 'paged mode'. Paged mode is turned on with **CTRL N** and off with **CTRL O**. When the computer is waiting at the bottom of a page both the shift lock and caps lock lights will be illuminated.

15 This code causes the computer to leave paged mode. See the previous entry (14) for more details.

16 This code (`VDU16` or **CTRL P**) clears the graphics area of the screen to the graphics background colour and the BASIC statement `CLG` has exactly the same effect. The graphics background colour starts off as black but may have been changed with the `GCOL` statement. `VDU16` does not move the graphics cursor – it just clears the graphics area of the screen.

17 `VDU17` is used to change the text foreground and background colours. In BASIC the statement `COLOUR` is used for an identical purpose. `VDU17` is followed by one number which determines the new colour. See the BASIC keyword `COLOUR` for more details.

18 This code allows the definition of the graphics foreground and background colours. It also specifies how the colour is to be placed on the screen. The colour can be plotted directly, ANDed, ORed or Exclusive-ORed with the colour already there, or the colour there can be inverted. In BASIC this is called `GCOL`.

The first byte specifies the mode of action as follows:

0   Plot the colour specified.
1   OR the specified colour with that already there.
2   AND the specified colour with that already there.
3   Exclusive-OR the specified colour with that already there.
4   Invert the colour already there.

The second byte defines the logical colour to be used in future. If the byte is greater than 127 then it defines the graphics background colour (modulo the number of colours available). If the byte is less than 128 then it defines the graphics foreground colour (modulo the number of colours available).

19 This code is used to select the actual colour that is to be displayed for each logical colour. The statements `COLOUR` (and `GCOL`) are used to select the logical colour that is to be text (and graphics) in the immediate future. However the actual colour can be redefined with `VDU19`. For example

```
MODE 5
COLOUR 1
```

will print all text in colour 1 which is red by default. However the addition of

`VDU 19,1,4,0,0,0` or `VDU 19,1,4;0;`
will set logical colour 1 to actual colour 4 (blue). The three zeros after the actual colour in the `VDU19` statement are for future expansion.

In `MODE 5` there are four colours (0, 1, 2 and 3). An attempt to set colour 4 will in fact set colour 0 so the statement

`VDU 19,4,4,0,0,0` or `VDU 19,4,4;0;`

is equivalent to

`VDU 19,0,4,0,0,0` or `VDU 19,0,4;0;`

We say that logical colours are reduced modulo the number of colours available in any particular `MODE`.

20 This code (`VDU20` or **CTRL T**) resets text and graphics foreground logical colours to their default values and also programs default logical to actual colour relationships. The default values are:

Two colour MODEs

0=Black
1=White

Four colour MODEs

0=Black
1=Red
2=Yellow
3=White

16 colour MODE

0=Black
1=Red
2=Green
3=Yellow
4=Blue
5=Magenta
6=Cyan
7=White
8=Flashing black/white
9=Flashing red/eyan
10=Flashing green/magenta
11=Flashing yellow/blue
12=Flashing blue/yellow
13=Flashing magenta/green
14=Flashing cyan/red
15=Flashing white/black

21 This code behaves in two different ways. If entered at the keyboard (as **CTRL U**) it can be used to delete the whole of the current line. It is used instead of pressing the **DELETE** key many times. If the code is generated from within a program by either **VDU21** or **PRINT CHR$(21);** it has the effect of stopping all further graphics or text output to the screen. The VDU is said to be disabled. It can be enabled with **VDU6**.

22 This VDU code is used to change **MODE**. It is followed by one number which is the new **MODE**. Thus **VDU22,7** is exactly equivalent to **MODE 7** (except that it does not change **HIMEM**).

23 This code is used to reprogram displayed characters. The ASCII code assigns code numbers for each displayed letter and number. The normal range of displayed characters includes all upper and lower case letters, numbers and punctuation marks as well as some special symbols. These characters occupy ASCII codes 32 to 126. If the user wishes to define his or her own characters or shapes then ASCII codes 224 to 255 are left available for this purpose. In fact you can redefine any character that is displayed, but extra memory must be set aside if this is done.

ASCII codes 0 to 31 are interpreted as VDU control codes – and this chapter is explaining the exact function of each. The full ASCII set consists of all the VDU control codes, all the normal printable characters and a user defined set of characters.

For example if the user wishes to define ASCII code 240 to be a small triangle then the following statement would have to be executed:

Character to be
redefined

VDU 23,240,1,3,7,15,31,63,127,255

Redefine character

Eight numbers giving the contents of each row of dots that makes up the desired character

$$= 1$$
$$2 + 1 = 3$$
$$4 + 2 + 1 = 7$$
$$8 + 4 + 2 + 1 = 15$$
$$16 + 8 \ldots = 31$$
$$= 63$$
$$= 127$$
$$= 255$$

Note that you cannot define your own characters in MODE 7.

See chapter 29 for a more detailed explanation.

As explained above the user may define any ASCII code in the range 224 to 255. To display the resultant shape on the screen the user can type

PRINT CHR$(240) or

VDU 240

In the unlikely event of the user wishing to define more than the 32 characters mentioned above (ASCII 224 to 255) it will be necessary to allocate more RAM for the purpose. This is described in chapter 43.

VDU23,1 can be used to turn the flashing cursor off:

VDU 23,1,0;0;0;0;

will turn the cursor off and

VDU 23,1,1;0;0;0;

will turn it on again.

A third use of **VDU23** is to permit the advanced programmer to alter the contents of the 6845 CRTC circuit. If the user wishes to place value X in register R this can be done with the command

```
VDU 23,0,R,X,0,0,0,0,0,0
```

The user is cautioned not to do this unless he or she understands how to program the 6845. Note however that when writing to register 7 (vertical synchronisation position) or register 8 (interlace) of the 6845, any offset that has been set up with the ***TV** statement (chapter 43) will be used to adjust the value sent to R7.

24 This code enables the user to define the graphics window – that is, the area of the screen inside which graphics can be drawn with the **DRAW** and **PLOT** statements. The graphics screen is addressed with the following coordinates.

Thus the coordinates of A would be approximately 1000,200.

When defining a graphics window four coordinates must be given; the left, bottom, right and top edges of the graphics area. Suppose that we wish to confine all graphics to the area shown below.

The left hand edge of the graphics area has an X value of (about) 150. The bottom of the area has a Y value of 300. The right hand side has X=1100 and the top has Y=700. The full statement to set this area is

```
VDU 24,150;300;1100;700;
```

Notice that the edges must be given in the order left X, bottom Y, right X, top Y and that when defining graphics windows the numbers must be followed by a semi-colon.

For those who wish to know why trailing semi-colons are used the reason is as follows: X and Y graphics coordinates have to be sent to the VDU software as two bytes since the values may well be greater than 255. The semi-colon punctuation in the **VDU** statement sends the number as a two byte pair with low byte first followed by the high byte.

25 This VDU code is identical to the BASIC **PLOT** statement. Only those writing machine code graphics will need to use it. **VDU25** is followed by five bytes. The first gives the value of K referred to in the explanation of **PLOT** in the BASIC keywords chapter. The next two bytes give the X coordinate and the last two bytes give the Y coordinate. Refer to the entry for **VDU24** for an explanation of the semi-colon syntax used.

For example

```
VDU 25,4,100;500;
```

would move to absolute position 100,500.

The above is completely equivalent to

```
VDU 25,4,100,0,244,1
```

26 The code **VDU26** **CTRL Z**) returns both the graphics and text windows to their initial values where they occupy the whole screen. This code repositions the text cursor at the top left of the screen, the graphics cursor at the bottom left and sets the graphics origin to the bottom left of the screen. In this state it is possible to write text and to draw graphics anywhere on the screen.

27 This code does nothing.

28 This code (**VDU28**) is used to set a text window. Initially it is possible to write text anywhere on the screen but establishing a text window enables the user to restrict all future text to a specific area of the screen. The format of the statement is

VDU 28,leftX,bottomY,rightX,topY

where   **leftX**   sets the left hand edge of the window
             bottomY   sets the bottom edge
             rightX   sets the right hand edge
             topY   sets the top edge

For the example shown the statement would be

VDU 28,5,20,30,12

Note that the units are character positions and the maximum values will depend on the **MODE** in use. The example above refers to **MODE1** and **MODE4**. In **MODES 2** and **5** the maximum values would be 19 for X and 24 for Y since these **MODE**s have only 20 characters per line.

29 This code is used to move the graphics origin. The statement **VDU29** is followed by two numbers giving the X and Y coordinates of the new origin. The graphics screen is addressed as shown below:

To move the origin to the centre of the screen the statement

```
VDU 29,640;400;
```

should be executed. Note that the X and Y values should be followed by semi-colons. See the entry for **VDU24** if you require an explanation of the trailing semi-colons. Note also that the graphics cursor is not affected by **VDU29**.

30 This code (**VDU30** or **CTRL ^**) moves the text cursor to the top left of the text area.

31 The code **VDU31** enables the text cursor to be moved to any character position on the screen. The statement **VDU31** is followed by two numbers which give the X and Y coordinates of the desired position.

To move the text cursor to the centre of the screen in **MODE 7** one would execute the statement

```
VDU 31,20,10
```

Note that the maximum values of X and Y depend on the **MODE** selected and that both X and Y are measured from the edges of the current text window not the edges of the screen.

32-126 These codes generate the full set of letters and numbers in the ASCII set. See the ASCII codes in the Appendices.

127 This code moves the text cursor back one character and deletes the character at that position. **VDU127** has exactly the same effect as the **DELETE** key.

128-223 These characters are normally undefined and will produce random shapes (see below and chapter 43).

224-255 These characters may be defined by the user using the statement **VDU23**. It is thus possible to have 32 user defined shapes such as

```
♣  VDU 23,224,8,28,28,107,127,107,8,28
♦  VDU 23,225,8,28,62,127,62,28,8,0
♥  VDU 23,226,54,127,127,127,62,28,8,0
♠  VDU 23,227,8,28,62,127,127,127,28,62
```

Note: You can use a ***FX** command which will then allow you to define characters 128 to 159 rather than 224 to 255. This has the advantage that you will then be able to use the new characters easily by holding down the **SHIFT** key while pressing one of the user definable (red) keys (see chapter 43).

35 Cassette files

This chapter summarises the facilities available for file handling using a cassette recorder. Refer to chapter 5 for an introduction to loading and saving BASIC programs.

Cassette motor control

Some cassette recorders have a 'remote' or 'automatic' motor control socket. This can be used with a switch on the microphone to start and stop the tape. If your recorder is of this type then the computer will be able to start and stop the tape automatically at the start and end of each BASIC program or section of recorded data.

If your cassette recorder does not have motor control then you will have to start and stop the tape manually. A light is provided on the keyboard to tell you when the tape should be running. This is labelled 'cassette motor'. When it is on, the tape should be running.

The description which follows assumes that you have automatic or remote motor control. If you don't then you'll have to start and stop the tape manually.

Recording levels

Many cassette recorders employ automatic record level. Recorders of this type do not have any 'record level' controls. If your recorder does not have automatic record level then you will have to set the record level yourself. Set the control so that the recording level indicator is slightly below the '0dB' level or the red mark.

Playback volume and tone

It is important that the playback volume is set correctly. You will need to experiment to find the correct level for your machine. The tone control should normally be set to 'maximum' or 'high'.

Keeping an index of programs

You will be able to record a large number of BASIC programs on a single cassette. However it is vital that the programs do not overlap on the tape. If they do then you will lose one of them. Beware of recording on the blank leader tape – always wind it on a little first.

If you forget what is on a tape then you can always use the command

*CAT

to obtain a 'catalogue' of the tape. When you give the command ***CAT** (and press the PLAY button on the recorder) the tape will play through, and the computer will print a catalogue of all the programs onto the screen. The catalogue gives the program name, the number of blocks (rather like pages in a book) used to record the program and lastly the length of the program (the number of letters in the book). It also checks that the recording is readable and reports any errors. As the catalogue is building up on the screen you will often see something like this

SKETCH   02

This indicates that the computer has found a file called **SKETCH** and that it is currently checking block 2 of that file. The block number is given in hexadecimal not decimal numbers. Press **ESCAPE** at the end of the tape to get back control of the computer.

Saving a BASIC program

A program that you have typed into the computer's memory can be saved onto cassette tape in the following way:

1. Insert the cassette into the recorder.

2. Type **SAVE"PROG"** and press **RETURN**.

PROG is just an example of a file name; file names are explained later in this chapter.

3. The message

RECORD then RETURN

will appear. Now use the fast forward and reverse buttons to position the tape at the correct place.

4. Press the RECORD and PLAY buttons on the cassette recorder.

5. Press the **RETURN** key on the computer to let it know that everything is now ready.

6. The computer will then record your program.

7. The tape will automatically stop when the computer has finished recording your program.

You can always abandon this process by pressing **ESCAPE**.

Saving a section of memory

This will not be needed by most people writing BASIC programs. It is most often used to record sections of machine code programs. The process is very similar to that employed to record a BASIC program.

1. Insert the cassette in the recorder.

2. Type

`*SAVE PROG SSSS FFFF EEEE RRRR`

and press **RETURN**.

3. Continue as for saving a BASIC program, above.

`SSSS` represents the start address of the data, in hexadecimal (hex).

`FFFF` represents the end address of the data plus one, in hex. As an option the format `+LLLL` can be used in this position. The plus sign is followed by the length of the data, in hex.

`EEEE` represents the (hex) execution address of the data. If the program is reloaded into the computer using the command

`*RUN PROG`

then once loaded the computer will jump to the specified execution address. The execution address is optional and if it is omitted the execution address will be assumed to be equal to the start address.

`RRRR` represents the (hex) reload address. This is optional, but if used the file will reload (using `*LOAD`, see below) at address `RRRR`. If `RRRR` is omitted then the file will reload at address `SSSS`.

Two examples may make the syntax clearer

`*SAVE patch 6000 6200`

`*SAVE match 4C00+0CE9 2A10 2000`

Loading a BASIC program

A BASIC program saved on cassette tape can be loaded into the computer's memory in the following way:

1. Insert the cassette in the recorder.

2. Type

`LOAD PROG`

and press **RETURN**.

3. The message

Searching

will appear.

4. Now use the fast forward and reverse buttons to position the tape at the correct place.

5. Press the PLAY button on the cassette recorder.

6. The computer will give you the message

Loading

when it finds the correct program. It will then load it into its memory.

7. The tape will automatically stop when the computer has finished loading.

When loading a program the usual catalogue-type display will appear. The message

Loading

will appear when the correct file is found. If the load should fail for any reason a message will appear.

Loading a machine code program

This will not be needed by most people using BASIC programs. It is used to load special purpose programs. The process is identical to that used to load a BASIC program except that the command is

*LOAD PROG AAAA

AAAA represents the absolute load address. It is optional but, if included, will force the program to load at the specified address. It therefore over-rides the reload address given when the program was saved. The program will load but not run; control will return to BASIC.

Two examples may make the syntax clearer:

*LOAD patch

100 MODE 7: *LOAD match 7E80

Loading and running a BASIC program

The statement **CHAIN** allows a BASIC program to **LOAD** and **RUN** another program. It is particularly useful when there is a sequence of related programs.

The command is used in exactly the same way as **LOAD** but with the word **CHAIN** substituted for the word **LOAD**.

1. Insert the cassette in the recorder.

2. Type `CHAIN PROG` and press **RETURN**.

3. The message

`Searching`

will appear.

4. Now use the fast forward and reverse buttons to position the tape at the correct place.

5. Press the PLAY button on the cassette recorder.

6. The computer will give you the message

`Loading`

when it finds the correct program. It will then load it into its memory.

7. The tape will automatically stop when the computer has finished loading and the computer will automatically run the program.

Loading and running a machine code program

A machine code program (not a BASIC program) can be loaded and run by using the statement

`*RUN PROG`

Using a cassette file to provide keyboard input

It is possible to get the computer to accept input from a cassette file instead of from the keyboard. In this case the cassette file would contain a set of commands, or answers to questions which a BASIC program would need. The command to force the computer to accept input from a file called `edit` would be:

`*EXEC edit`

File `edit` above is known as an EXEC file. EXEC files can contain BASIC commands or operating system commands (or both). Some operating system commands are listed in chapter 41. A comment line can be included in an EXEC file using a BASIC `REM` statement or by beginning the comment with `*|`.

To create a suitable cassette file you will need to use the BASIC statement `BPUT#` and not `PRINT#`, since the latter stores things in internal format. The command `*SPOOL` also creates suitable files – see chapter 37 for how to use it to merge programs.

Reading cassette data files

Data, as well as programs, may be recorded on cassette tape. This facility enables the user to keep records of names and addresses (for example) on tape for later use. Since the cassette tape can be started and stopped by the computer it also enables it to record results from experiments over many hours.

If the user wishes to read a data file then he or she must first open the file for input. In the process of opening a file the computer will allocate a channel number to the operation. If we wished to read in a list of names recorded on a data file called **NAMES** then the following statement would get the channel number into the variable **X**.

```
100 X=OPENIN("NAMES")
```

Once a file has been opened for input data can be read in from the tape. This can be done in two ways: a chunk at a time (for example a whole name) or a single letter at a time.

Data is read in a chunk at a time by using the **INPUT#X,A$** statement. This will read the first entry into **A$**.

To read a single character in use the function **A=BGET#(X)**.

Testing for end of file

While reading data in it is useful to test to see if the end of the file has been reached. This is done with the function **EOF#**. For example:

```
100 DIM B$(20)
110 Y=1
120 X=OPENIN("NAMES")
130 REPEAT
140 INPUT#X,A$
150 PRINT A$
160 B$(Y)=A$
165 Y=Y+1
170 UNTIL EOF#X
180 CLOSE#X
190 END
```

The program above reads up to 20 names off tape, prints them on the screen and then stores them in an array in memory. Of course if there were more than 20 names on file then the program would fail because the array can only hold 20 entries.

Storing data on tape

The data in the last example was read into an array in the computer. It could then be edited and the corrected version could be rerecorded on cassette.

The process of recording the data on the cassette consists of three steps: open the file for output, write out the data and then close the file. The example program records 20 entries back to tape from the array in memory.

```
200 X=OPENOUT("NEWNAMES")
210 FOR Y=1 TO 20
220 PRINT#X,A$(Y)
230 NEXT Y
240 CLOSE#X
250 END
```

Note that line 200 will make the computer issue the message

```
RECORD then RETURN
```

as it did when saving a BASIC program.

The `CLOSE#` statement will record any remaining data and then stop the recorder automatically.

Recording single characters on tape

Single characters (bytes) can be placed on tape using the command `BPUT#`. The following program stores the alphabet on tape. Note that the letter A has an ASCII value of 65 and the letter Z has an ASCII value of 90.

```
100 X=OPENOUT("ALPHABET")
110 FOR D=65 TO 90
120 BPUT#X,D
130 NEXT D
140 CLOSE#X
150 END
```

File names

File names on cassettes can be up to ten characters long and can include any character except space.

Responses to errors

If an error occurs during any of the following operations

```
SAVE
LOAD
CHAIN
*SAVE
*LOAD
*RUN
```

an error message and then the message

```
Rewind tape
```

will be printed on the screen. The user must then rewind the tape a short way and play it again. It is not usually necessary to replay the whole program, but only far enough to load the blocks causing errors.

If an error is detected during

```
BGET#
BPUT#
INPUT#
PRINT#
*EXEC
*SPOOL
```

the tape will stop and an error will be generated. This error can be trapped by BASIC in the usual way.

The user can escape at any time by pressing the **ESCAPE** key.

The error numbers generated by the cassette file system are as follows:

216 Data cyclic redundancy check (CRC) error.
217 Header CRC error.
218 An unexpected block number was encountered.
219 An unexpected file name was encountered.
220 Syntax error – for example an illegal ***OPT** statement.
222 An attempt was made to use a channel which was not opened.
223 End of file reached.

Changing responses to errors

If the user wishes to change the way the computer behaves when it detects an error during a cassette file operation (eg **LOAD** or **SAVE**) then this can be done with the ***OPT** statement.

*OPT by itself resets error handling and the 'message switch' to their default values. Default values are given later.

*OPT1,X is used to set the message switch which controls the detail of the message.

X=0 Implies no messages are issued.
X=1 Gives short messages.
X=3 Gives detailed information including load and execution addresses.

*OPT2,X is used to control the action that the computer takes when it detects an error during a cassette file operation.

X=0 Lets the computer ignore all errors though messages may be given.
X=1 The computer prompts the user to retry by rewinding the cassette.
X=2 The computer aborts the operation by generating an error.

*OPT3,X is used to set the inter-block gap used when recording data using **PRINT#** and **BPUT#**. The gap on **SAVE** is fixed. The value of X determines the gap in 1/10 seconds. If X is set to greater than 127 then the gap is set to the default value.

The default values for the ***OPT** command are:

For **LOAD**, **SAVE**, **CHAIN**, ***SAVE**, ***LOAD**, ***RUN**

*OPT,1   Short messages.

*OPT2,1   Prompt for retry.

*OPT3,6   0.6 second inter-block gap.

For **BGET#**, **INPUT#**, **BPUT#**, **PRINT#**

*OPT1,0   No messages.

*OPT2,2   Abort on error.

*OPT3,25   2.5 second inter-block gap.

Note: The effect of the ***OPT** command is different for each file system. When writing programs that are to run on cassette and disc the user must be fully aware of these different effects. The user is referred to the appropriate *Disc Filing System User Guide*.

Cassette tape format

The format of each block of data stored on cassette is given here for those whowish to produce tapes on other machines that may be read into the BBC Microcomputer.

– Five seconds of 2400Hz tone.
– One synchronisation byte (&2A).
– File name (one to ten characters).
– One end of file name marker byte (&00).
– Load address of file, four bytes, low byte first.
– Execution address of file, four bytes, low byte first.
– Block number, two bytes, low byte first.
– Data block length, two bytes, low byte first.
– Block flag, one byte.
– Spare, four bytes, currently &00.
– CRC on header, two bytes.
– Data, 0 to 256 bytes.
– CRC on data, two bytes.

Notes:
1. Each data block has its own header as shown above.
2. Load and execution addresses should have the top two bytes set to &FF in 8 bit machines.
3. Bit 7 of the block flag is set on the last block of a file.
4. CRC stands for cyclic redundancy check.

The header CRC acts on all bytes from the file name to the spare bytes inclusive. The data CRC acts on the data only. CRCs are stored high byte first and are calculated as follows. In the following C represents the character and H and L represent the high and low bytes of the CRC.

```
H=C EOR H
FOR X=1 TO 8
T=0
IF (bit 7 of H=1) THEN HL=HL EOR &0810:T=1
HL=(HL*2+T) AND &FFFF
NEXT X
```

The above algorithm is not a BASIC program!

36 Changing filing systems

The previous section dealt in detail with cassette files because the BBC Microcomputer is fitted with all the circuitry to enable cassette recorders to be used to store and to retrieve data.

Other filing systems can, of course, be used with the BBC Microcomputer. However, each of these will require additional hardware which will be accompanied by its own detailed instructions. Consequently, details are not given in this guide, for example, of the disc filing system. What is given below, however, is a list of the commands to enable the user to change from one filing system to another.

*TAPE	Selects the cassette filing system running at the default speed of 120 characters per second (1200 baud).
*TAPE3	Selects the cassette filing system running at 30 characters per second (300 baud).
*TAPE12	Selects the cassette filing system at 1200 baud.
*DISC	This statement selects the disc filing system so that all future file operations (eg **LOAD** and **SAVE**) work to the floppy disc units.
*ADFS	Selects the advanced disc filing system is.
*NET	Selects the Econet filing system for all future file operations.
*ROM	Selects the sideways ROM cartridge system.
*IEEE	The IEEE 488 interface filing system.
*TELESOFT	The Teletext filing system.

37 How to merge two BASIC programs

There are a number of ways of merging two BASIC programs. Two methods are given below which will work for both disc and cassette. The line numbers in the two programs should not clash unintentionally.

In order to merge two programs it is necessary to save one of them as an ASCII file rather than in the usual compressed format. This ASCII version of the program is then merged into the other program using the command ***EXEC**. Suppose we wish to merge the program **SHORT** into the program **LONG**. First, load in one of the programs

```
LOAD "SHORT"
```

and then create an ASCII version by typing in the next three lines

```
*SPOOL SHORT2
LIST
*SPOOL
```

This produces an ASCII version of the program. The ASCII version is called **SHORT2**.

Now load in the big program by typing

```
LOAD "LONG"
```

and finally merge in the small program by typing

```
*EXEC SHORT2
```

The command

```
*SPOOL SHORT2
```

informs the computer that anything that it outputs to the screen is also to be sent to a file called **SHORT2**. When the computer lists the file it therefore creates an ASCII listing. The command ***SPOOL** without a file name is used to end or close the spooled file.

Having created the ASCII version of **SHORT** called **SHORT2**, the user then loads the file **LONG**. The command ***EXEC "SHORT2"** tells the computer to read in the file **SHORT2** as if it were getting characters from the keyboard. If the file **SHORT2** contains line numbers (as it does), then these lines will just be added to the BASIC program. Of course, a line number read in from **SHORT2** will replace any line with an identical line number in **LONG**, so it is necessary to renumber the two programs **SHORT** and **LONG** so that their line numbers don't clash.

A quicker method is given below – but if you use this method you must make sure that the second program that you loaded in uses larger line numbers than the first program. You will get surprising results if not.

Firstly load the program (with the lower line numbers) in the normal way. Then get the computer to print, in hex, the top-of-program address less two by entering

```
PRINT ~TOP-2
```

Call the resultant number **XXXX**. Then enter

```
*LOAD SHORT XXXX
```

to load the program **SHORT** (or whatever you have called it) in above the first program. Finally type **END** to get the computer to sort its internal pointers out. A typical dialog might look like this – assuming that one program is called **ONE** and the other is called **TWO**.

```
>LOAD "ONE"
>PRINT~TOP-2
       195F
>*LOAD TWO 195F
>END
```

This method is very easy, but you must look after the line numbers.

38 Using printers

The BBC Microcomputer can be used with the vast majority of printers available today. It is not possible, though, to drive old 'Teletypes' which operate at ten characters per second, or printers requiring a 20mA current loop connection. To use a printer with your BBC Microcomputer you will need to do three things:

– Connect the printer to the computer.
– Tell the computer whether your printer is plugged into the parallel or serial printer port.
– Tell the computer to copy everything sent to the screen to the printer.

Connect the printer to the computer

As the above implies there are two possible sockets (ports) to which you can connect the printer. The *parallel printer port* is often called a Centronics compatible port. Printers that need to be connected to parallel ports usually have a 36-way Amphenol socket. To connect the printer to the computer you will need a cable made up as follows from a 26-way insulation displacement connector, about one metre of 26-way ribbon cable and one 36-way Amphenol plug.

The 26-way cable should be inserted into the 36-way plug so that pins 1 to 14 and 19 to 32 are connected. The cable should be inserted into the 26-way plug so that pin 1 of the Amphenol 36-way plug is connected to pin 1 of the 26-way insulation displacement plug.

36-way
Amphenol plug

26-way
ribbon cable

Bump

Notch

1

26-way insulation
displacement plug

26

A parallel printer cable

Both plugs viewed looking at the pins on the underside of the plug body.

Parallel printer connections

Name	36-way plug pin number	26-way plug pin number
Strobe	1	1
Data 0	2	3
Data 1	3	5
Data 2	4	7
Data 3	5	9
Data 4	6	11
Data 5	7	13
Data 6	8	15
Data 7	9	17
Acknowledge	10	19
Ground	19 to 32	All even numbers (except 26)

The *serial printer port* is often referred to as an RS232 or V24 port. The standard connector plug is a Cannon 25-way D type and there is also a standard for the connections to the plug. Predictably however, different manufacturers disagree on the interpretation of the standard! The serial port on the BBC Microcomputer emerges via a 5-pin DIN plug and it will normally be possible to drive a serial printer with just three wires. The views of the two plugs are shown below from the outside of the case and looking into the respective sockets.

5-way DIN plug
for computer

25-way D type plug for printer

Proceed as follows:

– Connect a wire between GND and pin 7.
– Connect another wire between data out and pin 3.
– The position of the third wire varies from printer to printer.
– For Qume printers connect CTS to pin 20.
– If that doesn't work then try CTS to pin 4.

If you require a more complete RS232 connection try the following:

– Data in to pin 2 (transmit data).
– Data out to pin 3 (receive data).
– CTS to pin 4 (clear to send).
– RTS to pin 5 (request to send).
– 5 GND to pin 7 (signal ground).

Telling the computer whether you are using a serial or parallel printer

*FX5 is used to inform the computer whether it is to send printer output to the parallel or serial port. Type:

*FX5,1   if you are using a parallel printer

*FX5,2   if you are using a serial printer

*FX5,4   if you are using the Econet printer

The computer defaults to (assumes unless you tell it otherwise, that you are using) a parallel printer.

If you have selected a serial printer then you will have to set the baud rate to match that on the printer. The default setting is 9600 baud but many printers run at other rates. Select one of

*FX 8,1   75 baud

*FX 8,2   150 baud

*FX 8,3   300 baud

*FX 8,4   1200 baud

*FX 8,5   2400 baud

*FX 8,6   4800 baud

*FX 8,7   9600 baud

*FX 8,8   19200 baud (this rate is not guaranteed)

Telling the computer to copy everything to the printer

This is often called turning the printer on – but, of course, we are not referring to switching the printer on but to enabling and disabling its printing.

To turn it on type **CTRL B** and to turn it off type **CTRL C**.

In a BASIC program **VDU2** will turn the printer on and **VDU3** will turn it off.

Note that all output will now appear on the screen and printer. To send output to the printer only you can use ***FX3**.

For disc systems, ***TYPE "filename"** will print out a data file.

Characters not sent to the printer

Some printers automatically move the paper up one line when they receive a carriage return. Since the computer also moves the paper up one line (by using a line feed), the paper may move up two lines instead of one. It is possible to tell the computer not to send any particular character to the printer. The most common requirement is to ignore line feed which has an ASCII code of 10.

*FX6,10 will set it to ignore line feeds. ***FX6** should be set after ***FX5** has been used to select the printer type. The default is to ignore line feeds.

The complete set-up sequence for a serial printer running at 1200 baud and which does not require line feeds would be:

*FX 5,2   Serial printer.

*FX 8,4   1200 baud.

*FX 6,10   No line feeds (default so it may be omitted).

A few more detailed points:

Data is transmitted using one start bit, eight data bits and one stop bit.

Receive baud rates may be the same as, or may differ from, transmit baud rates. Receive baud rates are set by ***FX7**.

The user may supply his or her own specialist printer driver routine. The address of the user routine should be placed at location &222 and the user defined routine can be selected as the printer output routine with ***FX5,3**.

If the user intercepts the operating system write character routine (OSWRCH) then all the VDU control codes must be dealt with. When a BASIC program executes **DRAW 10,10** a string of six bytes is sent to the VDU driver via OSWRCH. In this case the bytes would be **25,5,10,0,10,0,** so beware!

39 Indirection operators

Indirection operators are not normally available on personal computers. They enable the user to read and write to memory in a far more flexible way than by using PEEK and POKE. They are intended for programmers using the computer's assembler or programmers wishing to create their own data structures. There are three indirection operators:

Name	Purpose	Number of bytes affected
? query	Byte indirection operator	1
! shriek	Word indirection operator	4
$ dollar	String indirection operator	1 to 256

For the sake of illustration let us play with the memory around location &2000 (that is 2000 hex – an appreciation of hex numbers is essential. If you don't understand hex then you will not need to use indirection operators). Let us set the variable **M** to &2000

```
M= &2000
```

?M means the contents of memory location **M**, so to set the contents of &2000 to 40 we write

```
?M = 40
```

and to print the contents of memory location &2000 we write

```
PRINT ?M
```

Those familiar with other dialects of BASIC will realise that

Y=PEEK(X) becomes **Y=?X** and
POKE X,Y becomes **?X=Y**
The query operator acts on one byte of memory only. The word indirection operator (shriek) acts on four successive bytes. For example

```
!M=&12345678
```

would set location

```
&2000=&78
&2001=&56
&2002=&34
&2003=&12
```

and **PRINT ~!M** (print in hex, shriek M) would give

```
12345678
```

Four bytes are used to store integers so the shriek operator can be used to manipulate integer variables.

The last operator in this group is the string indirection operator called dollar. Do not confuse **$M** with **M$** as they are quite different. **M$** is the usual string variable. On the other hand **$M** can be used to read or write a string to memory starting at memory location **M**. Remembering that we have set

M = &2000 then

$M="ABCDEF" will place the string **ABCDEF** and a carriage return (&0D) in memory from location &2000 on.

Similarly **PRINT $M** will print

```
ABCDEF
```

And one last twist to the use of these operators. We have seen how query, shriek and dollar can be used as unary operators – that is with only one operand. We have used **M** as the operand in the example above – for example

```
M=&2000
?M=&45
Y=?M
PRINT ?M

!M=&8A124603
Y=!M
PRINT!M

$M="HELLO ZAPHOD"
B$=$M
PRINT $M
```

but in addition both query and shriek can be used as binary operators provided that the left hand operand is a variable (such as **M9**) and not a constant.

Thus **M?3** means the contents of (memory location M plus 3) – in other words of location &2003.

There is a simple routine to examine the contents of memory for 12 bytes beyond &2000.

```
10 FOR X=0 TO 12
20 PRINT ~M+X, ~M?X
30 NEXT
```

Line 20 reads 'print in hex (M plus X) and in the, next column, in hex, the contents of (M plus X)'. It is easy to write this into one of the user defined function keys and keep it for debugging – like this

```
*KEY 0 FOR X=0 TO 12: P. ~M+X, ~M?X:NEXT |M
```

but don't forget that in **MODE 7** it will be displayed as

```
*KEY 0 FOR X=0 TO 12: P. ÷M+X, ÷M?X:NEXT‖M
```

just to complicate matters!

Here are some illustrations of some of the above.

Set up function key 0 and use it to examine memory beyond &2000.

```
>*KEY 0 FOR X=0 TO 12:P. ~M+X, ~M?X:N.|M

>M=&2000

>FOR X=0 TO 12:P. ~M+X, ~M?X:N.
          2000              FF
          2001              FF
          2002              FF
          2003              FF
          2004              FF
          2005              FF
          2006              FF
          2007              FF
          2008              FF
          2009              FF
          200A              FF
          200B              FF
          200C              FF
```

Use the byte indirection operator to change one byte.

```
>M?3=&33
>FOR X=0 TO 12:P. ~M+X, ~M?X:N.
          2000                FF
          2001                FF
          2002                FF
          2003                33
          2004                FF
          2005                FF
          2006                FF
          2007                FF
```

```
           2008            FF
           2009            FF
           200A            FF
           200B            FF
           200C            FF
```

Use the word indirection operator to change four bytes.

```
>M!2=&12345678
>FOR X=0 TO 12:P. ~M+X, ~M?X:N.
           2000            FF
           2001            FF
           2002            78
           2003            56
           2004            34
           2005            12
           2006            FF
           2007            FF
           2008            FF
           2009            FF
           200A            FF
           200B            FF
           200C            FF
```

Use the string indirection operator to insert a string into a known place in memory.

```
>$M="ABCDEFG"
>FOR X=0 TO 12:P. ~M+X, ~M?X:N.
           2000            41
           2001            42
           2002            43
           2003            44
           2004            45
           2005            46
           2006            47
           2007             D
           2008            FF
           2009            FF
           200A            FF
           200B            FF
           200C            FF
```

Note that interesting structures can be generated using `!` and `$`. For example you may need a structure containing a ten character string, a number and a reference to a similar structure. `!` and `$` together can do this. If `M` is the address of the start of the structure then

`$M`   is the string

`M!11` is the number

`M!15` is the address of the next structure

The tools are therefore provided to enable you to manipulate general tree structures and lists very easily in BASIC.

40 HIMEM, LOMEM, TOP and PAGE

These four pseudo-variables give the programmer an indication of the way the computer has allocated the available memory. **PAGE** and **TOP** give the bottom and top of the user's program so

PRINT TOP-PAGE

can be used to find out how big a program is.

HIMEM gives the top of memory so

PRINT HIMEM-TOP

will indicate how much space is left.

When you run the program the computer will need some space to store variables so you cannot use up all the available memory just with your program.

Random Access Memory extends from location 0 to location 32767 (in hexadecimal that is &7FFF). RAM is normally allocated in **MODE 7** as shown in the diagram. As the user enters more program the program grows, increasing the value of **TOP**. Normally the computer stores program variables immediately above the user program but this can be changed by resetting the variable **LOMEM**.

Again the computer normally expects to be able to use all the memory up to that set aside for the screen, but the user may move the position of the highest boundary by changing **HIMEM**.

Note that in the 'shadow screen' mode, **HIMEM** always returns a value of &8000 (see chapter 42 for more details).

The variable **TOP** is calculated by the computer on request. It does this by starting at **PAGE** and working through the program. The user cannot reset the value of **TOP** but can reset **PAGE**, **HIMEM** and **LOMEM** if needed for some special purpose. On a cassette system **PAGE** will be set to &0E00. On a disc system (DFS) **PAGE** would be &1900. On a system with an Econet interface (NFS) **PAGE** would be &1200. On a system with both disc and Econet interfaces fitted **PAGE** will be set to &1B00. If the Advanced Disc Filing System (ADFS) fitted, **PAGE** is &1D00. ADFS with DFS or NFS gives a **PAGE** value of &1F00, and ADFS with DFS and NFS gives a **PAGE** value of &2100.

The above notes, and the diagram, refer only to machines without a second processor fitted.

41 Operating system statements

The BBC Microcomputer includes a large and powerful operating system. This can be accessed from user written machine code routines or from BASIC. If a BASIC statement starts with an asterisk then the whole of the rest of the statement is passed to the operating system directly. Note that the filing systems listed below can only be accessed if your BBC Microcomputer is fitted with the appropriate hardware. Full details of each filing system are contained in their related user guides. The operating system (OS) commands include:

*LOAD	Loads a section of memory (not a BASIC program) (see chapter 35).
*SAVE	Saves a section of memory (see chapter 35).
*RUN	Loads and executes a machine code program (see chapter 35).
*CAT	Displays a catalogue of files on the cassette, disc or Econet disc unit (see chapter 35).
*KEY	Programs one of the user defined keys (see chapters 25 and 43).
*OPT	Determines how the computer reacts to errors during loading of cassettes and the amount of detail given during cassette operations.
*FX	Enables the user to control a large number of the computer effects such as flash rate (see chapter 43).
*TAPE	Selects the cassette filing system running at 1200 baud.
*TAPE3	Selects the cassette filing system running at 300 baud.
*DISC	This statement selects the disc filing system so that all future file operations (eg **LOAD** and **SAVE**) work to the floppy disc units.
*ADFS	Selects the advanced disc filing system for all future file operations.
*NET	Selects the Econet filing system for all future file operations.

*SPOOL	Copies all screen output to a named file (see chapter 37).
*EXEC	Uses a named file to provide input as if it had been typed in at the keyboard (see chapters 35 and 37).
*MOTOR	Used to turn the cassette motor on or off.
*TV	This can be used to move the whole of the displayed picture up or down the screen and also controls the picture interlace (see chapter 43). The interlace on the BBC Microcomputer is off by default.
*ROM	Selects the sideways ROM filing system.
*SHADOW	Selects the shadow screen on a subsequent **MODE** change (see chapter 42).
*IEEE	Selects the IEEE 488 interface filing system.
*TELESOFT	Selects the Teletext filing system.

All these OS statements may, if required, have the contents of BASIC variables passed to them as parameters by using OSCLI (see chapter 33).

42 The shadow screen

There are two areas of memory in the computer that can be selected for use as display memory: from &3000 (variable, according to the display **MODE** selected) to &8000 in the main memory map, or an area outside the main memory map known as the 'shadow RAM'. Using the shadow RAM leaves the whole of user memory (ie from **PAGE** to &8000) free for your use.

Display **MODE**s **128** to **135** use the shadow RAM as the display memory by default, and are the shadow screen equivalents of **MODE**s **0** to **7** (respectively). To enter a shadow display **MODE**, simply type in its **MODE** number – for example:

MODE 135 RETURN

would enter the shadow screen equivalent of **MODE 7**. Non-shadow mode may be re-entered by changing to a non-shadow **MODE**. The following command sequence illustrates the memory map changes which take place when switching between shadow and non-shadow **MODE**s:

```
MODE 3

P. ~HIMEM
      4000
MODE 131

P. ~HIMEM
      8000
MODE3

P. ~HIMEM
      4000
```

(The screen would be cleared by each **MODE** change).

To force the operating system to enter shadow mode on subsequent **MODE** changes, (even if a non-shadow **MODE** is selected) type

*SHADOW RETURN

To return to non-shadow mode type

*SHADOW 1 RETURN

followed by a **MODE** change to a non-shadow **MODE**. The following command sequence illustrates the memory map changes which take place when using the ***SHADOW** commands:

```
MODE 3

P. ~HIMEM
     4000
*SHADOW
P. ~HIMEM
     4000
MODE 3

P. ~HIMEM
     8000
```

MODE 131 would have the same effect as the second **MODE 3** command above, although in that case there would be no need for the preceding ***SHADOW** command. Continuing from the above sequence we have:

```
*SHADOW 1
P. ~HIMEM
     8000
MODE 3

P. ~HIMEM
     4000
```

Substituting **MODE 131** for **MODE 3** above would give:

```
P. ~HIMEM
     8000
```

The ***SHADOW** commands are useful when switching between software which uses shadow RAM and software which doesn't.

Other shadow mode-related commands

*FX114 produces the same effect as ***SHADOW**, and ***FX114,1** produces the same effect as ***SHADOW 1**.

VDU22,<128+n> (used when in non-shadow mode) will select **MODE** n; shadow mode will not be entered and **HIMEM** will not be reset.

Shadow mode is retained across a soft break, but is reset to non-shadow by a hard break.

43 The operating system and how to make use of it

What is the operating system?

The operating system (or the machine operating system (MOS) to give it its full title) is a program which controls the operation of the computer. The MOS for the BBC Microcomputer lives in ROM, and is activated automatically as soon as the microcomputer is switched on. The MOS runs constantly, performing repetitive tasks with such speed that you are not even aware that anything is going on; indeed, the MOS has already done a tremendous amount of work before you even press a key! The functions performed by the MOS include scanning the keyboard, updating the screen, performing analogue-to-digital conversions (for input from joysticks) and controlling the sound generator.The MOS is a large and complex machine code program (more about machine code later), consisting of a number of routines which perform functions such as those listed above. Like all computer programs, the MOS will not do anything until it is told to (although as already mentioned, just switching the computer on, and leaving it on, tells the MOS to do quite a lot). All the functions performed by the MOS can be made available to other machine code programs, and one machine code program which you will be familiar with is BASIC. BASIC provides an easy way of using the microcomputer's computing power, and it makes constant use of the MOS routines to get input from the keyboard and reflect that input on the screen. How ever, the degree of control over the MOS provided by BASIC is limited. For example, typing

`COLOUR 9` **RETURN**

in `MODE 2` will cause subsequent screen display to appear in flashing red/cyan, the 'work' being done by the MOS routines. However, BASIC provides no way of varying the flash rate. There are three principal ways of achieving fine control of this kind:

1. Using `*FX` commands

2. Calling operating system routines from BASIC

3. Calling operating system routines from assembly language

Although the rest of this chapter covers all of the above three topics, you are referred to the next chapter for a detailed introduction to assembly language.

The *FX commands

The ***FX** commands are a family of operating system commands which act directly on the MOS, causing its routines to carry out a variety of functions which are summarised below.

The general form of a ***FX** command is

*FXa,x,y

where a, x and y are integers. (There can be a space between the **x** and the a.) For example, the MOS version number can be printed out on the screen by typing

*FX0 **RETURN**

Note that in the above command a=0, but x and y are absent. Another example, (which incidentally solves the 'flashing colour' problem which was referred to earlier) is

*FX9,10

which causes the first named colour (red in our example) to be displayed for ten fiftieths of a second (***FX9,20** would display it for twenty fiftieths of a second, and so on). Note here that the default duration of the first named colour – ie the time for which it is displayed before a ***FX9** command is typed in to change the time – is half a second, so restoring the default duration could be achieved by typing

*FX9,25 **RETURN**

Incidentally, loading the default duration (plus many other default parameter values) into RAM is one of the jobs done by the MOS when the microcomputer is switched on. Typing

*FX9,0 **RETURN**

would set the duration to infinity, causing the first colour to be displayed continuously, and the second colour not to be displayed at all. Exactly the same effect could be achieved by typing

*FX9 **RETURN**

which illustrates the fact that if the x and y parameters are omitted they are taken to be equal to zero. As a final example,

*FX144,2,1

would move the display two lines up the screen and turn the interlace off (after the next **MODE** change or soft break). A summary of all the ***FX** commands and their functions is given later in this chapter.

All the ***FX** commands can be typed in in immediate mode or can be included as a line of a BASIC program. The values typed in for x and y (if any) are copied by the MOS into registers known as X and Y, which reside in the microcomputer's 6512 microprocessor (see next chapter for more details). Some ***FX** commands return values in X and Y, but unfortunately the MOS does not copy these returned values back into X% and Y% (it doesn't even 'know' about X% and Y%). However, this drawback can be overcome through the use of a certain operating system call. An operating system call is a means of activating one of the routines which exist within the MOS. Several operating system calls exist, which perform a variety of functions (see chapter 45). The operating system call which we will examine first is OSBYTE.

OSBYTE calls from BASIC

OSBYTE has call address &FFF4, a memory location within the operating system ROM. This means that when OSBYTE is called, the instruction held in location &FFF4 is executed. This instruction loads an address which is held in a RAM location whose address is given in locations &FFF5 and &FFF6 (&020A for OSBYTE). OSBYTE is said to 'indirect' via &20A. Locations &20A and &20B contain the address of (or the 'vector' to) the routine within the MOS which is actually executed. This may seem a little complicated, but it does mean (for example) that OSBYTE's call address will always be &FFF4, and its indirection vector will always be at &20A, no matter how much the MOS changes between different versions.

OSBYTE can be called from within BASIC in two ways, one of these being with the **CALL** BASIC keyword. Try typing the following sequence:

```
A%=0
X%=0
Y%=0
CALL(&FFF4)
```

You will notice that this has exactly the same effect as the ***FX0** command. This is because a ***FX** command and an OSBYTE call are very nearly one and the same thing: the MOS recognises any command beginning ***FX** as an OSBYTE call. in general,

```
*Fxa,x,y
```

is equivalent to OSBYTE with A=a, X=x and Y=y. A is another register in the 6512 (known as the 'accumulator', see chapter 45) whose contents are set equal to the value read by the BASIC interpreter from the resident integer variable A%. As another example, the following program would have the same effect as ***FX9,10** (see above):

```
10 MODE2
20 COLOUR 9
30 A%=9
40 X%=10:Y%=0
50 CALL(&FFF4)
```

Obviously, this is far more ponderous than simply typing ***FX9,10**, and indeed the **CALL** BASIC keyword is mainly used in conjunction with assembly language programs (see chapter 45). A far more powerful method of making operating system calls is through use of the **USR** BASIC keyword. The above program could be re-written as

```
10 MODE2
20 COLOUR 7
30 A%=9
40 X%=10:Y%=0
50 PRINT ~USR(&FFF4)
```

(Note that the foreground colour has been changed to white – it's a little difficult to read the output of this program in flashing red/cyan!). The effect of line 50 is to set the duration of the first flashing colour as before (which can be confirmed by using **COLOUR** to select a flashing colour combination), and also to print the following on the screen:

30191909

The string of figures are hexadecimal digits, which have come from registers in the 6512. They have the following meanings (reading from right to left):

09   This has come from register A (the accumulator), which was set up from A% (it is simply the OSBYTE call number; all OSBYTE calls preserve A).

19   This has come from the X register, and is the previous colour duration value. In this case the previous value is the default value, 25. (Remember the digit pairs are hexadecimal, &19=25 decimal).

19   This has come from the Y register, and contains the same value as the X register.

30   This has come from the P register (the program status register, mainly of interest to those programming in assembly language).

Of course, all OSBYTE calls do not have the same exit conditions. The next example performs a totally different function. Note also that the result of the **USR** call has been assigned to a set of variables, so that it can be printed out in a more readable form:

```
 10 REM Uses OSBYTE with A=129 to wait one second
for a key
 15 REM to be pressed.
 20 DIM registers 4
 30 DIM A$(2):A$(0)="A=":A$(1)="X=":A$(2)="Y="
 40 A%=129
 50 X%=100
 60 Y%=0
 70 !registers=USR(&FFF4)
 80 FOR K=0 TO 2
 90 PRINT A$(K);~registers?K
100 NEXT K
```

This program uses OSBYTE with A=129 to read a key, which must be pressed within a specified time limit (one second in the example). If it reminds you of BASIC's **INKEY** keyword that's because **INKEY** uses the same OSBYTE call. If a time limit of **n** centiseconds is required then X and Y should be set up as follows:

```
X=n AND &FF
Y=n DIV &100
```

The exit conditions vary according to what happens; if no key is pressed within the time limit, X is preserved, Y=&FF. If a key is pressed, X returns the hexadecimal ASCII code of the key character, Y=0.

All the ***FX** calls and OSBYTE calls available on the BBC Microcomputer are detailed at the end of this chapter, preceded by a numerical and a functional summary. Other calls are available to various filing systems which may not be fitted to your machine; these are noted, but you are referred to the appropriate filing system user guide for details.

OSBYTE calls from assembly language

This section does not attempt to provide an introduction to assembly language and how to use it; you are referred to the next chapter for that. For those unfamiliar with the concept of assembly language, it is safe to assume for now that it is a sort of 'half way house' between BASIC and machine code, which is what the 6512 microprocessor actually executes. Programs written in assembly language are more difficult to write than BASIC programs, but they usually take up less memory and are faster than an equivalent BASIC program, which is why large, complex programs such as arcade games will always be written in assembly language.

Shown below is an assembly language equivalent of the 'read a key within one second' example which we have already seen in BASIC

```
 10 OSBYTE=&FFF4
 20 P%=&3500
 30 [
 40 LDA#129
 50 LDX#100
 60 LDY#0
 70 JSR OSBYTE
 80 RTS
 90 ]
100 CALL &3500
```

The first thing you will notice about the above program is that lines 10, 20 and 100 seem to be BASIC statements. Indeed they are, the program is really a 'hybrid'. Line 30 tells BASIC 'assembly language starts here', line 90 says 'assembly language finished'. Line 40 says 'load the accumulator with decimal 129'. Similarly, lines 50 and 60 load the X register with decimal 100 and the Y register with 0. Line 70 calls OSBYTE, although the machine code assembled from lines 40 to 80 is not actually executed until the program reaches line 100. This description of the program is deliberately brief, but any questions that you may have will be answered in the next chapter. Try typing the program in and running it. You will see that six lines of text appear on the screen (more about that in the next chapter), there is a pause, and then the BASIC prompt reappears. If you run the program again, but press a key immediately after pressing **RETURN**, you will find that the same thing happens, except that the prompt reappears when you press the key. In both cases the program doesn't appear to have done anything! In fact, it has carried out exactly the same processing as the BASIC example, but it hasn't printed any results out on the screen. This illustrates a disadvantage of making an operating system call from assembly language: there is no direct way of printing out the contents of the 6512 registers. (This isn't as bad as it sounds because real-life applications of assembly language programs, arcade games for example, rarely require register contents to be printed.) The following program gets around this problem (albeit in a limited and rather unsatisfactory way) by making use of another MOS routine, called OSWRCH:

```
 10 OSBYTE=&FFF4
 20 OSWRCH=&FFEE
 30 P%=&3500
 40 [
 50 LDA#129
 60 LDX#100
 70 LDY#0
```

```
 80 JSR OSBYTE
 90 TXA
100 JSR OSWRCH
110 RTS
120 ]
130 CALL &3500
```

Of interest here is line 90, which turns an exit parameter from OSBYTE (in the X register) into an entry parameter for OSWRCH (supplied in the A register). Try running the program. If you don't press a key after **RETURN**, then the first character on the last line to be printed out is **d**, which has decimal ASCII code 100, which was the value loaded into the X register. Pressing a key will cause the key character to be displayed followed immediately by the the BASIC prompt, indicating that the program has finished. What has happened is that OSWRCH has read the value in A, transformed it into a character and printed the character on the screen (without a line feed or carriage return). More details of OSWRCH and all the other operating system calls are given in chapter 45.

The foregoing has probably left you somewhat unimpressed by the potentialities of assembly language programming, but hopefully the next chapter will change your views!

The *FX commands and OSBYTE calls

This section lists and details all the most useful ***FX** commands and OSBYTE calls. There are others which are recognised by the operating system but there is not sufficient space to document them here. You are referred to the many other publications which are available concerning the BBC Microcomputer for descriptions of these other, somewhat esoteric calls. The decimal and hexadecimal calls which appear in the following tables are the values of a for ***FX**a, or the values to be loaded into the accumulator for the related OSBYTE call.

Functional summary (alphabetical)

heading	brief description	dec	hex
buffer	flush selected class	15	F
	flush selected buffer	21	15
	get buffer status	128	80
	insert value into buffer	138	8A
	get character from buffer	145	91
	examine buffer status	152	98
bus	read from FRED	146	92
	write to FRED	147	93
	read from JIM	148	94
	write to JIM	149	95
	read from SHEILA	150	96
	write to SHEILA	151	97
cursor	enable/disable cursor edit keys	4	4
	read text cursor position	134	86
	read character at cursor	135	87
ESCAPE	clear ESCAPE condition	124	7C
	set ESCAPE condition	125	7D
	acknowledge ESCAPE	126	7E
events	disable event	13	D
	enable event	14	E
files	close ***SPOOL** and ***EXEC** files	119	77
	check end of file status	127	7F
input/output	select input device	2	2
	select output device(s)	3	3
keyboard	auto-repeat delay	11	B
	auto-repeat rate	12	C
	read **CTRL/SHIFT** key status	118	76
memory	explode soft character RAM allocation	20	14
	select shadow/non-shadow display memory	114	72
	read high order address	130	82
	read OSHWM address	131	83
	read bottom of display RAM address	132	84
	read lowest address for particular **MODE**	133	85

operating system	print version number	0	0
paged ROM	enter language ROM	142	8E
printer	select printer type	5	5
	set printer ignore character	6	6
	end of user print routine	123	7B
RS423	receive baud rate	7	7
	transmit baud rate	8	8
serial	enable/disable 6850 ACIA IRQ	232	E8
soft keys	enable/disable cursor edit keys	4	4
	reset soft keys	18	12
	cancel VDU queue	224	E0
	set base number for function key codes	225	E1
	set base number for **SHIFT** function key codes	226	E2
	set base number for **CTRL** function key codes	227	E3
	set base number for **SHIFT CTRL** function key codes	228	E4
sound	sound on/off	210	D2
speech	read from speech processor	158	9E
	write to speech processor	159	9F
	speech on/off	209	D1
	return presence of speech processor	235	EB
user	user OSBYTE call	1	1
VDU	read VDU status	117	75
VIA/6522	enable/disable user 6522 IRQ	231	E7
	enable/disable system 6522 IRQ	233	E9
video	set flash period of first colour	9	9
	set flash period of second colour	10	A
	wait for field synchronisation	19	13

Numerical summary

Decimal	Hex	Function
0	0	Prints operating system version number.
1	1	Reserved for application programs.
2	2	Selects input device.
3	3	Selects output devices.
4	4	Enable/disable cursor edit keys.
5	5	Select printer type.
6	6	Set printer ignore character.
7	7	Set RS423 receive baud rate.
8	8	Set RS423 transmit baud rate.
9	9	Set flash period of first colour.
10	A	Set flash period of second colour.
11	B	Set auto-repeat delay.
12	C	Set auto-repeat period.
13	D	Disable various events.
14	E	Enable various events.
15	F	Clear all or just input buffer.
16	10	Select number of ADC channels.
17	11	Force start of conversion on ADC channel.
18	12	Reset user defined function keys.
19	13	Wait for field synchronisation.
20	14	Explode soft character RAM allocation.
21	15	Clear selected buffer.
114	72	Control shadow/main memory selection
117	75	Read VDU status byte.
118	76	Read **CTRL/SHIFT** key status.
119	77	Close ***SPOOL** and ***EXEC** files.
123	7B	End of user print routine.
124	7C	Reset ESCAPE flag.
125	7D	Set ESCAPE flag.
126	7E	Acknowledge detection of escape condition.
127	7F	Check end of file status.
128	80	Read ADC channel/fire buttons/last conversion.
129	81	Read key within time limit.
130	82	Read machine high order address.
131	83	Read top of operating system RAM address.
132	84	Read bottom of display RAM address.
133	85	Read lowest address for particular **MODE**.
134	86	Read text cursor position.
135	87	Read character at text cursor position.

137	89	Turn cassette motor on/off.
138	8A	Insert character into specified buffer.
139	8B	Set file options.
140	8C	Select cassette file system and set speed.
142	8E	Select sideways ROM.
144	90	Alter TV display position/interlace.
145	91	Remove character from buffer.
146	92	Read from I/O area FRED.
147	93	Write to I/O area FRED.
148	94	Read from I/O area JIM.
149	95	Write to I/O area JIM.
150	96	Read from I/O area SHEILA.
151	97	Write to I/O area SHEILA.
152	98	Examine specified buffer.
158	9E	Read from speech processor.
159	9F	Write to speech processor.
209	D1	Speech on/off.
210	D2	Sound on/off.
218	DA	Read/write size of VDU queue
239	EF	Read/write shadow display mode state
224	E0	Cancel VDU queue.
225	E1	Set base number for function key codes.
226	E2	Set base number for **SHIFT** function key codes.
227	E3	Set base number for **CTRL** function key codes.
228	E4	Set base number for **SHIFT CTRL** function key codes.
229	E5	**ESCAPE**=&1B.
230	E6	Enable/disable normal **ESCAPE** key action.
231	E7	Enable/disable user 6522 IRQ.
232	E8	Enable/disable 6850 ACIA IRQ.
233	E9	Enable/disable system 6522 IRQ.
235	EB	Return presence of speech processor.
253	FD	Last reset type.
255	FF	Write start-up option byte.

Each ***FX** and OSBYTE call is now explained in detail.

*FX0 prints a message giving the version number of the operating system.

*FX1,n sets the user flag to n. See OSBYTE with A=1 for more details.

*FX2 selects the input device from which characters will be retrieved.

*FX2,0 gets characters from the keyboard and disables the RS423 receiver.

*FX2,1 gets characters from the RS423 port, and disables the keyboard.

*FX2,2 gets characters from the keyboard and enables the RS423 receiver.

*FX3 is used to select whether output appears (or not) on different output streams. ***FX3** is followed by one number, eg ***FX3,4**. The second number should be considered in its binary form, because the value of each bit determines the effect of the command. Here is a list of the bits involved, and the effect of each bit when its value is 1.

bit 0 enable RS423 driver

bit 1 disable VDU driver

bit 2 disable printer driver completely

bit 3 enable printer driver (as long as bit2 at 0)

bit 4 disable **SPOOL**d output

bit 5 not used

bit 6 disable printer driver unless character preceded by **CTRL** A or **VDU1** (as long as bit1 and bit2 at 0)

bit 7 not used

Here are some useful values in the table below

	Printer	**Screen**	**RS423**
*FX3,0	enabled	on	off
*FX3,1	enabled	on	on
*FX3,2	enabled	off	off
*FX3,3	enabled	off	on
*FX3,4	off	on	off
*FX3,5	off	on	on
*FX3,6	off	off	off
*FX3,7	off	off	on
*FX3,8	on	on	off
*FX3,9	on	on	on
*FX3,10	on	off	off
*FX3,11	on	off	on
*FX3,12	off	on	off
*FX3,13	off	on	on
*FX3,14	off	off	off
*FX3,15	off	off	on

*FX3,0 to ***FX3,3** 'enable' the printer, which means that **VDU2** (or **CTRL** B) will send output to printer. Values from 0 to 3, 8 to 12, 16 to 19 etc are affected by ***FX5**.

Values from 16 to 31 are equivalent to values from 0 to 15, except that ***SPOOL** output is turned off.

*FX4,0 resets the system so that the cursor editing keys and **COPY** perform their normal cursor editing function.

*FX4,1 disables the cursor editing and makes the cursor editing keys generate normal ASCII codes. The codes are shown below in both decimal and hex numbers.

COPY	135	&87
←	136	&88
→	137	&89
↓	138	&8A
↑	139	&8B

*FX4,2 permits the and **COPY** keys to be programmed in the same way as the user defined function keys. In other words they may contain strings. In this case the ***KEY** numbers are

COPY	11
←	12
→	13
↓	14
↑	15

*FX5 is used to select the printer type.

*FX5,1 selects output to the parallel (Centronics type) output connector.

*FX5,2 selects the serial RS423 output.

*FX5,3 selects a user supplied printer driver (see chapter 38).

Note that none of these will actually cause output to appear on the printer. **VDU2** (or **CTRL B**) must be used to start output going to the selected printer channel.

*FX5,0 can be used to select a 'printer sink' where characters can be lost without the possibility of the system 'hanging' with a full printer buffer.

*FX6 is used to set the character which will be suppressed by the printer driver routines. Some printers do an automatic 'line feed' when they receive a 'carriage return'. It is therefore useful to be able to prevent the line feed characters from reaching the printer. The ASCII code for line feed is decimal code 10 so this can be achieved with the statement

*FX6,10

Having set this mode it is still possible to get a line feed through to the printer by use of the **VDU1** (send next character to printer only) statement. The character following **VDU1** is not checked, thus to send a line feed one would use

```
VDU1,10
```

*FX6 should always be executed after the printer type has been set by ***FX5**.

*FX6,0 provides no character filtering.

Note that the default is that line feed characters are filtered out.

*FX7 is used to select the baud rate to be used when receiving data on the RS423 interface, as follows:

*FX7,1   75 baud receive

*FX7,2   150 baud receive

*FX7,3   300 baud receive

*FX7,4   1200 baud receive

*FX7,5   2400 baud receive

*FX7,6   4800 baud receive

*FX7,7   9600 baud receive

*FX7,8   19200 baud receive (this rate is not guaranteed)

*FX8 is used to select the transmit rate to be used on the RS423 interface.

*FX8,1   75 baud transmit

*FX8,2   150 baud transmit

*FX8,3   300 baud transmit

*FX8,4   1200 baud transmit

*FX8,5   2400 baud transmit

*FX8,6   4800 baud transmit

*FX8,7   9600 baud transmit

*FX8,8   19200 baud transmit (this rate is not guaranteed)

Note: The standard receive/transmit format adopted for RS423 is one start bit, eight bits, one stop bit.

*FX9 and ***FX10** are used to set the flash rate of flashing colours. Colours always flash between one colour and its complement. Thus in **MODE 2**, **COLOUR 9** selects flashing red/cyan.(See chapter 34 for more information on **COLOUR**.)

*FX9 is followed by a number which gives the duration of the first named colour. The duration is given in fiftieths of a second. Thus to select one fifth of a second one would use the statement

*FX9,10

The default is ***FX9,25**

*FX10 is used to set the period of the second named colour. See the entry above for ***FX9**.

Note that values of 0 for the duration will eventually force the selected state to occur full time. If neither counter is set to zero then setting the other counter to zero will immediately force the latter colour to appear.

The default is ***FX10,25**.

*FX11 If a key is held depressed then after a short delay (the auto-repeat delay) the key will repeat automatically every so often. The delay is given in hundredths of a second.

*FX11 sets the delay before the auto-repeat starts.

*FX11,0 turns off the auto-repeat all together.

*FX11,5 would set the auto-repeat delay to five hundredths of a second, etc.

*FX12 sets the auto-repeat period, that is the amount of time between the generation of each character.

*FX12,0 resets both the auto-repeat delay and the auto-repeat to their normal values.

*FX12,10 would set the auto-repeat period to ten hundredths of a second thus producing ten characters per second.

*FX13 and ***FX14** are used to disable and enable certain events. These ***FX** calls will only be used by programmers writing in Assembly Language and more information on events is given in chapter 44. The summary which follows should be read in conjunction with that chapter.

*FX13,0 disables output buffer empty event.

*FX13,1 disables input buffer full event.

*FX13,2 disables character entering input buffer event.

*FX13,3 disables ADC conversion complete event.

*FX13,4 disables start of vertical synchronisation event.

*FX13,5 disables interval timer crossing zero event.

*FX13,6 disables **ESCAPE** pressed event.

*FX13,7 disables RS423 receive error event.

*FX13,8 disables service/Econet error event.

*FX14,0 enables output buffer empty event.

*FX14,1 enables input buffer full event.

*FX14,2 enables character entering input buffer event.

*FX14,3 enables ADC conversion complete event.

*FX14,4 enables start of vertical synchronisation event.

*FX14,5 enables interval timer crossing zero event.

*FX14,6 enables **ESCAPE** pressed event.

*FX14,7 enables RS423 receive error event.

*FX14,8 enables service/Econet error event.

The initial condition is that all are disabled.

*FX15 enables various internal buffers to be cleared (or emptied).

*FX15,0 clears all internal buffers.

*FX15,1 clears the currently selected input buffer.

See also ***FX21**.

As an OSBYTE call this typically executes in 370 microseconds.

*FX16 is used to select the number of analogue to digital converter (ADC) channels that are to be used. Each ADC channel takes 10ms to convert an analogue voltage into a digital value. Thus if all four ADC channels are enabled then new values for a particular channel will only be available every 40ms. It is, therefore, often wise to enable only the number of channels actually required.

*FX16,0 will disable all ADC channels.

*FX16,1 will enable ADC channel 1 only.

*FX16,2 will enable channels 1 and 2.

*FX16,3 will enable channels 1, 2 and 3.

*FX16,4 will enable all four ADC channels.

*FX17 forces the ADC converter to start a conversion on the selected channel. Thus ***FX17,1** will start a conversion on channel 1. Channels are numbered 1, 2, 3 and 4. OSBYTE call with A=&80 can be used to determine when the conversion is complete as can **ADVAL(0)**.

*FX18 resets the user defined function keys so that they no longer contain character strings.

*FX19 causes the machine to wait until the start of the next frame of the display for animation.

*FX20 when the machine starts up, space is allocated at &C00 for the redefinition of 32 displayed characters using the **VDU23** statement.

In the initial state, or after issuing a ***FX20,0** command, the character definitions are said to be 'imploded'. This means that an attempt to print any character with ASCII code greater than 128 (&80) will be mapped on to the characters stored at memory &C00. Initially this will produce garbage. Once a user defined character has been defined, the effect of the mapping is to produce the same character for four ASCII codes, each code being offset from its neighbour by 32 (decimal). Therefore, for example, ASCII codes &80, &A0, &C0 and &E0 will all show the same character. Redefining any one of the above codes would also identically redefine the other three. Note that code &80 (or &A0. &C0 or &E0) will be mapped to memory location &C00; code &81 (or &A1, C1 or &E1) will mapped to memory location &C08, etc. Character codes in the range &20 to &7F can also be redefined, but will map (in the ***FX20,0** state) to the same memory locations, thus overwriting what is already there.

After a ***FX20,6** command the character definitions are said to be fully 'exploded' and all printing characters (codes &20 to &FF) can be redefined, each group of 32 characters in the range having its own memory allocation (see below). However, until any character in a block is redefined, the block will map on to either the preset ROM (&20 to &7F) or on to memory at &C00 (&80 to &FF). The memory allocated to each block of ASCII codes in the fully exploded state is:

ASCII code	Memory allocation
&20 to &3F	OSHWM + &300 to OSHWM + &3FF
&40 to &5F	OSHWM + &400 to OSHWM + &4FF
&60 to &7F	OSHWM + &500 to OSHWM + &5FF
&80 to &9F	&C00 to &CFF
&A0 to &BF	OSHWM to OSHWM + &FF
&C0 to &DF	OSHWM + &100 to OSHWM + &1FF
&E0 to &FF	OSHWM + &200 to OSHWM + &2FF

The character definitions may also be partially exploded by using a parameter between 0 and 6. For example, **`*FX20,1`** will allow character codes &80 to &BF to be redefined. Each increment in the parameter allocates another page (256 bytes) of memory above OSHWM to the soft character definitions. **`*FX20,4`** will allow redefinition of character codes &80 to &FF and &20 to &3F.

The following command sequence should serve to make the situation clearer. (Press **RETURN** after each line.)

```
VDU23, &E2,1,3,7,15,31,63,127,255
*FX20,3
VDU23,&E2,255,127,63,31,15,7,3,1
VDU&E2
*FX20,0
VDU&E2
```

Although the fourth and sixth lines invoke the same character code, a different character is displayed each time due to the effect of the **`*FX20`** commands. (Note that **`*FX20`** could be typed in instead of **`*FX20,0`**.)

If this is done, and the programmer redefines characters in the range &20 to &3F, he or she must leave memory up to OSHWM + &3FF clear of his or her program. This is done by first finding the value of OSHWM (for the particular configuration in use) by using OSBYTE call 131 and then setting **PAGE** to a value &400 above this value. Of course, if the programmer only wishes to redefine ASCII codes in the range &80 to &9F (128 to 159) then he or she need not alter **PAGE**.

Codes in the range &80 to &9F are of particular importance since the programmer can generate them directly from the keyboard by holding down **SHIFT** at the same time as pressing one of the user definable function keys.

See later in this chapter for more information on Operating System High Water Mark.

`*FX21` allows any internal buffer to be cleared (or flushed).

`*FX21,0` flushes the keyboard buffer.

`*FX21,1` flushes the RS423 serial input buffer.

`*FX21,2` flushes the RS423 serial output buffer.

`*FX21,3` flushes the printer output buffer.

`*FX21,4` flushes the sound channel number 0 (noise).

`*FX21,5` flushes the sound channel number 1.

`*FX21,6` flushes the sound channel number 2.

*FX21,7 flushes the sound channel number 3.

*FX21,8 flushes the speech synthesis buffer.

See also ***FX15**, OSBYTE &80 and OSBYTE &8A.

*FX114 is used to select shadow/main display memory at the next **MODE** change. ***FX 114** selects shadow memory, ***FX114,1** selects main memory. Note that a **MODE** change is required before **HIMEM** will be reset. (See chapter 42 for more details of the shadow screen facility.)

*FX119 causes any open files being used as ***SPOOL**ed output or ***EXEC**ed input to be closed.

*FX124 clears the **ESCAPE** pressed flag (bit 7 of location &FF).

*FX125 sets the **ESCAPE** pressed flag. It must be set with this call and not directly. It has a similar effect to pressing the **ESCAPE** key. (Conditional on any OSBYTE 229 call.)

*FX126 must be used to acknowledge the detection of an escape condition if the user is intercepting characters fed into the computer through the OSRDCH (Operating System Read Character) routine.

*FX142 can be used to select one of the six sideways ROM sockets (note that only language ROMs can be selected). The sockets are numbered,counting anticlockwise from the MOS/BASIC socket at the top right, 14/15, 10/11, 8/9, 2/3, 4/5, and 6/7. Each socket is capable of accepting a 32K ROM. The paired socket numbers correspond to the two 16K 'logical ROMs' which may exist within the single 32K 'physical ROM'. If a socket contains a 16K ROM then it can be selected by using either of the two socket numbers. For example, a (16K) language ROM in the socket immediately to the left of the MOS/BASIC socket could be selected by typing ***FX142,10** or ***FX142,11**.

*FX209 will turn speech on or off. ***FX209,80** is on, ***FX209,1** is off.

*FX210 will turn the sound on or off. ***FX210,0** is on, ***FX210,1** is off.

*FX218,0,0 will cause the VDU software to forget the bytes it has received so far as part of an (incomplete) VDU (with parameters) statement.

*FX255 will set the keyboard start-up option byte. This value will only take effect on a soft reset. Any other sort of reset (hard or power-on) and the value will be taken direct from the wiring on the keyboard. The links on the keyboard are all 'unmade', which ensures that the RAM location which can be accessed by this call (&28F) has all its bits set on start-up or by a hard break. The value contained in &28F and the corresponding keyboard link numbers are as follows:

Links 1 and 2 (bits 7 and 6 respectively) are unused.

Links 3 and 4 (bits 5 and 4 respectively) set the disc drive timings. The default settings give the *slowest* disc access – this ensures that the computer can be used with any make of disc drive.

Link 5 (bit 3) selects auto or manual boot on (default setting gives manual boot).

Links 6, 7, and 8 (bits 2, 1, and 0) select the screen **MODE** (default settings give **MODE 7**).

For example, ***FX255,203** would give fast discs, manual boot, and **MODE 3** on soft reset.

OSBYTE calls

OSBYTE call with A=&0 (0) Read operating system version number

If X=0 on entry then the operating system version number will be returned as an error message. If X is non-zero on entry then on exit X will contain the opeating system series number – for example OS 2.00 would return X=2. On exit A and Y are preserved, C is undefined.

OSBYTE call with A=&1 (1) Read/write the user flag

On entry, Y=0 for a write operation or Y=&FF for a read operation. For a write operation, X=new value of user flag. On exit, X=previous user flag value, Y=&FF.

This call uses OSBYTE with A=&F1 (241), and is not used by the operating system, being left free for user applications. The user flag is stored at location &281 and has default value 0.

OSBYTE call with A=&2 (2) Select input device

On entry, the value in X determines the input device(s), as follows:

X=0   Keyboard selected, RS423 disabled.
X=1   RS423 selected, keyboard disabled.
X=3   Keyboard selected, RS423 enabled (but not selected.)

On exit, X=0 if previous input was from the keyboard, X=1 if previous input was from the RS423. A is preserved, Y an C are undefined.

OSBYTE call with A=&3 (3) Select output device

On entry, the value in X determines the output device to be selected, as follows:

Bit 0 set   – Enables RS423 driver.
Bit 1 set   – Disables VDU driver.
Bit 2 set   – Disables printer driver.
Bit 3 set   – Enables printer, independently of **CTRL B** or **CTRL C**.

Bit 4 set	– Disables spooled output.
Bit 5	– Not used.
Bit 6 set	– Disables printer driver unless the output character is preceded by a `VDU1` command (or equivalent).
Bit 7	– Not used.

The default options (X=0) are:

RS423 disabled
VDU enabled
Printer enabled (if selected by `VDU2`)
Spooled output enabled (if selected by `*SPOOL`)

This OSBYTE call uses OSBYTE with A=&EC (236). On exit, A is preserved, X contains the previous `*FX3` status, Y and C are undefined.

OSBYTE call with with A=&4 (4) Enable/disable cursor editing keys

On entry, the value in X determines the editing keys' status, as follows:

X=0	Cursor editing enabled (default setting).
X=1	Cursor editing disabled. The cursor control keys will return normal ASCII codes, see `*FX4,1` (later in this chapter) for details.
X=2	Cursor editing disabled. The cursor control keys act as soft keys with soft key association numbers as detailed under `*FX4,2` (see later in this chapter).

On exit, A is preserved, X contains the previous `*FX4` setting, Y and C are undefined.

OSBYTE call with A=&5 (5) Select printer type/output channel

On entry, the value in X determines the print destination, as follows:

X=0	Printer sink (printer output ignored).
X=1	Parallel printer (default setting).
X=2	Serial printer (RS423 output). This setting will produce the effect of a printer sink if the RS423 is enabled using OSBYTE with A=3.
X=3	User supplied printer driver routine, the address of which should be placed starting at location &222.
X=4	Econet printer.
X=5-255	User supplied printer driver routine (see X=3 above).

On exit, A is preserved, X contains the previous `*FX5` setting, Y and C are undefined. This call enables interrupts, and is not reset to default by a soft break.

OSBYTE call with A=&6 (6) Select character to be ignored by printer

On entry, X contains the decimal ASCII code of the character to be ignored. The effect of this call can be suppressed through the use of an appropriate **VDU1** statement. The default setting is X=10 (line feed). On exit, A is preserved, X contains the previous ***FX6** setting, Y and C are undefined.

OSBYTE call with A=&7 (7) Set RS423 receive baud rate

On entry, the value in X determines the baud receive rate, as follows:

X=1	75 baud
X=2	150 baud
X=3	300 baud
X=4	1200 baud
X=5	2400 baud
X=6	4800 baud
X=7	9600 baud
X=8	19200 baud (this rate not guaranteed)

On exit A is preserved, X and Y contain the old serial ULA register contents, C is undefined.

OSBYTE call with A=&8 (8) Set RS423 transmit baud rate

On entry, the value in X determines the baud transmit rate, as follows:

X=1	75 baud
X=2	150 baud
X=3	300 baud
X=4	1200 baud
X=5	2400 baud
X=6	4800 baud
X=7	9600 baud
X=8	19200 baud (this rate not guaranteed)

On exit A is preserved, X and Y contain the old serial ULA register contents, C is undefined.

OSBYTE call with A=&9 (9) Set flash rate of flashing colours (first colour)

On entry, the value in X determines the duration in centiseconds of the first named colour, X=0 sets the duration to infinity; X=25 is the default setting. On exit, A is preserved, X and Y contain the value of the previous duration setting, C is undefined.

OSBYTE call with A=&A (10) Set flash rate of flashing colours (second colour)

On entry, the value in X determines the duration in centiseconds of the second named colour. X=0 sets the duration to infinity; X=25 is the default setting.

On exit, A is preserved, X and Y contain the value of the previous duration setting, C is undefined.

OSBYTE call with A=&B (11) Set keyboard auto-repeat delay

On entry, the value in X determines the delay in centiseconds before auto-repeating starts. X=0 disables the auto-repeat facility; X=50 is the default value. On exit, A is preserved, X contains the previous setting, Y and C are undefined.

OSBYTE call with A=&C (12) Set keyboard auto-repeat rate

On entry, the value in X determines the auto-repeat periodic interval in centiseconds. X=0 resets the auto-repeat delay and rate to their default values; X=8 is the default value. On exit, A is preserved, X contains the previous setting, Y and C are undefined.

OSBYTE call with A=&D (13) Disable events

On entry, X contains the event code, corresponding to n for the ***FX13,**n commands (see later in this chapter). Note that disabling an event means that a user-supplied event handling routine will *not* be activated should the disabled event occur. The default state is all events disabled. See chapter 44 for more information on events. On exit, A is preserved, X and Y contain the old enable state (0=disabled), C is undefined.

OSBYTE call with A=&E (14) Enable events

On entry, X contains the event code, corresponding to n for the ***FX14,**n commands (see later in this chapter). Note that enabling an event means that a user-supplied event handling routine *will* be activated should the enabled event occur. The default state is all events disabled. See chapter 44 for more information on events. On exit, A is preserved, X and Y contain the old enable state (>0=disabled), C is undefined.

OSBYTE call with A=&F (15) Flush selected buffer class

On entry, the value in X determines the class of buffer to be flushed, as follows:

X=0 All buffers flushed
X<>0 Input buffer flushed only

On exit, the buffer contents are discarded, A is preserved, X, Y and C are undefined.

OSBYTE call with A=&10 (16) Select ADC channels which are to be sampled

On entry, X contains the number of channels to be sampled, corresponding to n for the ***FX16,**n commands (see later in this chapter). If n=0, sampling is disabled, if n>4 then n is reset to 4. On exit, A is preserved, X contains the previous setting, Y and C are undefined.

OSBYTE call with A=&11 (17) Force ADC conversion on specified channel

On entry, X contains the required channel number. If X>4 then X is reset to 4. See also OSBYTE with A=&80 (128). On exit, A is preserved, X is preserved unless it is greater than 4 – in which case X is reset to 4, Y and C are undefined.

OSBYTE call with A=&12 (18) Reset soft keys

This call resets the user-defined function keys so that they no longer contain character strings. There are no entry conditions. On exit, A and Y are preserved, X and C are undefined.

OSBYTE call with A=&13 (19) Wait for field synchronisation

This call causes the machine to wait until the start of the next frame of the display. This occurs 50 times per second and can be used for timing or animation. User trapping of IRQ1 (see chapter 44) may stop this call from working. There are no entry conditions. On exit, A and Y are preserved, X and C are undefined.

OSBYTE call with A=&14 (20) Explode soft character RAM allocation

This call assigns memory blocks for use by soft character defenitions. On entry, the value in X determines the memory block(s) to be assigned, corresponding to n for the `*FX20,`n commands (see later in this chapter). See also OSBYTE call with A=&B6 (182). On exit, A is preserved, X contains the new OSHWM (high byte), Y and C are undefined.

OSBYTE call with A=&15 (21) Flush specified buffer

On entry, the value in X determines the buffer to be flushed, corresponding to n for the `*FX21,`n commands (see later in this chapter). See also OSBYTE calls with A=&0F (15) and &80 128). On exit, A and X are preserved, Y and C are undefined.

OSBYTE call with A=&72 (114) Control shadow/main memory selection

On entry, X=0 selects the shadow display RAM, X=1 selects the main display RAM. Note that the value of `HIMEM` is not reset by this command. On exit, X contains the previous display memory selection, A is preserved, Y and C are undefined. (See chapter 42 for details of the shadow screen facility).

OSBYTE call with A=&75 (117) Read VDU status byte

This call returns the VDU status byte (which contains various status flags) in the X register. The bits in X are as follows:

— Bit 0 set if **VDU2** sent. Cleared by **VDU3**.
— Bit 2 set if paged mode on. Cleared if paged mode off.
— Bit 3 set if software scrolling. Cleared if hardware scrolling. (Software scrolling is used when text windows have been defined whereas hardware scrolls are used when the whole screen scrolls.)
— Bit 4 set if shadow display mode selected.
— Bit 5 set when cursors are joined by **VDU5**.
— Bit 7 set if VDU disabled by **VDU21**.

OSBYTE call with A=&76 (118) Read CTRL/SHIFT key status

This call returns with the carry bit set if the **CTRL** key is pressed, and with the negative bit set if the **SHIFT** key is pressed. Machine code routines may branch on these conditions.

OSBYTE call with A=&77 (119) Close any SPOOL or EXEC files

This call closes any open files being used as ***SPOOL**ed output or ***EXEC**ed input to be closed. This call also performs a paged ROM call with A=&10. On exit, A is preserved, X, Y and C are undefined.

OSBYTE call with A=&7B (123) End of user print routine

This call is used by the user print routine to indicate to the MOS that it has finished its task. (Cancels a ***FX5,3** command.)

OSBYTE call with A=&7C (124) Reset ESCAPE flag

This call clears any ESCAPE condition without any further action (ie no buffers are flushed and no open EXEC files are closed). The Tube (if it is active) is informed. The ESCAPE flag is stored as the top bit of location &FF, and should never be interfered with directly. This OSBYTE call can only be usefully entered from within an assembly language program, since BASIC itself resets the ESCAPE flag. There are no entry conditions; on exit, A, X and Y are preserved, C is undefined.

OSBYTE call with A=&7D (125) Simulate ESCAPE condition

This call partially simulates the **ESCAPE** key being pressed (conditional on any OSBYTE 229 call). The Tube (if it is active) is informed. An ESCAPE event is not generated. Note that if this OSBYTE call is made from BASIC, BASIC will reset the ESCAPE flag (see OSBYTE call A=125). There are no entry conditions; on exit, A, X and Y are preserved, C is undefined.

OSBYTE call with A=&7E (126) Acknowledge detection of an ESCAPE condition

This call attempts to acknowledge the existence of an ESCAPE condition. It should be used if you are using OSRDCH (see chapter 44) to intercept characters being fed into the computer, and it is also useful as part of a user-supplied ESCAPE condition service routine. If an ESCAPE condition is detected, all active buffers will be flushed and any open EXEC files will be closed. There are no entry conditions; on exit, X=255 if the ESCAPE condition existed (X=0 otherwise), A is preserved, Y and C are undefined.

OSBYTE call with A=&7F (127) (EOF#)

This call returns the end of file status of a file which has been previously opened. On entry X should contain the channel number allocated to the file. On exit X will be zero if the end of file has not been reached and X will be non-zero if the end d

OSBYTE call with A=&80 (128) Read ADC channel (ADVAL)

This call returns the most recent value of a particular analogue to digital converter channel. It can also be used to detect an end of conversion and to see if the games fire buttons are pressed.

On entry X contains the channel number to be read. If X is in the range 1 to 4 then the specified channel will be read and on exit the 16 bit value will be returned in X and Y. Y will contain the eight most significant bits and X the least significant bits.

If on entry X=0 then on exit Y will contain a number in the range 0 to 4 indicating which channel was the last to complete. Note that ***FX16** and ***FX17** reset this value to 0. A value of zero indicates that no channel has completed. Also on exit the two least significant bits of X will indicate the status of the two fire buttons. The user should always AND X with 3 to mask out high order bits.

If on entry X contains a negative number (in twos complement notation) then the call will provide information about various input buffers. On entry Y must contain &FF.

X on entry	Buffer checked
255	Keyboard buffer
254	RS423 serial input buffer
253	RS423 serial output buffer
252	Printer output buffer
251	Sound channel 0 (noise)
250	Sound channel 1
249	Sound channel 2
248	Sound channel 3
247	Speech buffer

On exit X contains a number giving, for input buffers, the number of characters in the buffers. For output buffers X contains the number of spaces still free in the buffer.

OSBYTE call with A = &81 (129) Read key within time limit

The call waits for a character from the current input channel until a time limit expires or it tests a particular key. The BASIC function **INKEY** uses this call. The programmer is reminded that this call will immediately obtain characters if the user has 'typed ahead'. It is therefore often necessary to flush the input buffer with ***FX15,1** before using this call.

The maximum time delay is passed to the subroutine in X and Y. The delay is measured in hundredths of a second and Y contains the most significant byte and X the least significant byte. The maximum delay is &7FFF hundredths of a second which is about five and a half minutes.

On exit, Y=0 if a character was detected within the time limit. In this case X contains the character. Y=&1B indicates that **ESCAPE** was pressed. This must be acknowledged with ***FX126**. Y=&FF indicates a time-out.

If on entry Y contains a negative number then the routine can be used to check for a specific key closure. See the BASIC keyword **INKEY** for more information.

As an OSBYTE call a 'check for character waiting with zero delay' takes typically 130 microseconds.

OSBYTE call with A=&82 (130) Read machine high order address

The BBC Microcomputer uses a 6512 processor (a close relative of the 6502) which requires a 16 bit address. However a number of routines require a 32 bit address – for example most file system addresses are 32 bits wide to ensure compatibility with future products. A specific high order address – that is the top 16 bits of the total 32 bit address – is allocated to the present BBC Microcomputer. The high order address is returned in X and Y with Y containing the most significant byte and X the least significant byte.

OSBYTE call with A=&83 (131) Read top of OS RAM address (OSHWM)

The Machine Operating System uses memory from page zero up to store operating system variables. The exact amount of RAM needed depends, for example, on whether or not the disc operating system is in use.

This call is used to return the address of the first free location in memory above that required for the operating system. The value is returned in X and Y with Y containing the most significant byte and X containing the least significant byte.

```
10A%=131
30PRINT ~USR(&FFF4)
>RUN
B10E0083
```

`&0E00` is the value returned, `&83` is the OSBYTE code.

OSBYTE call with A=&84 (132) Read bottom of display RAM address

This call returns, in X and Y, the lowest memory address used by the screen display or by special paged ROMs. It indicates the top of free RAM that BASIC can safely use. `HIMEM` is normally set to this value when using the `MODE` statement. As usual, Y contains the most significant byte and X the least significant byte of the result. If this call is made following a `*SHADOW` command, &8000 will only be returned following a `MODE` change. (See chapter 42 for details of the shadow screen facility).

OSBYTE call with A=&85 (133) Read lowest address for particular MODE

This call returns, in X and Y, the address of the start of memory that would be set aside if a particular display `MODE` were to be selected. Certain paged ROMs might also affect the value returned. The display `MODE` to be investigated is passed in X. This call does not change `MODE` s, it merely investigates the possible consequences of doing so. If this call is made following a *SHADOW command, it will return &8000 without the need for a `MODE` change. (See chapter 42 for details of the shadow screen facility).

OSBYTE call with A=&86 (134) Read text cursor position

This call returns in X and Y the X and Y coordinates of the text cursor. A similar function is performed in BASIC by `POS` and `VPOS`. As an OSBYTE call thistakes typically 100 microseconds.

OSBYTE call with A=&87 (135) Read character at text cursor position

On exit X will contain the character at the text cursor's position and Y will contain a value representing the current graphics display **MODE** . If the character cannot be recognised then X will contain zero.

The following function could be used to read the character at position X,Y in a BASIC program.

```
2000 DEF FNREADCH(X,Y)
2010 LOCAL A%, LASTX, LASTY,C
2020 LASTX=POS
2030 LASTY=VPOS
2040 VDU 31,X,Y
2050 A%=135
2060 C=USR(&FFF4)
2070 C=C AND &FFFF
2080 C=C DIV &100
2090 VDU 31, LASTX, LASTY
2100 = CHR$(C)
```

The call takes typically 120 microseconds.

OSBYTE call with A=&89 (137) Motor control

This call is similar to the ***MOTOR** statement in BASIC. If only one cassette recorder is in use then setting

X=0 will turn the motor off
X=1 will turn the motor on

The cassette filing system (CFS) controls the motor using this OSBYTE call and sets Y as follows:

Y=0 for write operations
Y=1 for read operations

As a result the user can easily implement a dual cassette system by trapping any OSBYTE call with A=&89 and activating (via his or her own hardware) the second recorder for, say, all write operations. The normal internal motor control could then be activated for all read operations.

OSBYTE call with A=&8A (138) Insert character into specified buffer

This enables characters to be inserted into any buffer. On entry X contains the buffer number and Y contains the character to be inserted. Buffer numbers are as follows:

0   Keyboard buffer
1   RS423 serial input buffer
2   RS423 serial output buffer
3   Printer output buffer
4   Sound channel 0 buffer
5   Sound channel 1 buffer
6   Sound channel 2 buffer
7   Sound channel 3 buffer
8   Speech buffer

Therefore to place the letter R (ASCII code 82) into the keyboard input buffer you would use

```
*FX138,0,82
```

In machine code the X register must contain the buffer number and the Y register the character, eg

```
10 DIM GAP 20
20 OSBYTE = &FFF4
30 P%=GAP
40 [
50 LDA #138
60 LDX #0
70 LDY #82
80 JSR OSBYTE
90 RTS
100]
110
120 CALL GAP
>RUN
1B82
1B82  A9  8A    LDA #138
1B84  A2  00    LDX #0
1B86  A0  52    LDY #82
1B88  20  F4 FF JSR OSBYTE
1B8B  60        RTS
```

(see next chapter for more details of machine code and assembly language.)

OSBYTE call with A=&8B (139) File options

This call is directly equivalent to `*OPT` and it controls the computer's response to errors during file operations such as `LOAD` and `SAVE`. See chapter 35 for more details.

For example, with the cassette filing system `*FX139,1,0` would issue no messages during file operations.

`*FX139,2,2` would make the computer abort if any error were detected.

`*FX139,3,5` would result in 0.5 second inter-block gaps.

On entry X contains the option number and Y contains the particular selected option. Thus

```
LDA #139
LDX #1
LDY #0
JSR &FFF4
```

would ensure that no error messages were issued during file operations.

OSBYTE call with A=&8C (140) Tape speed

This call is directly equivalent to `*TAPE` which selects the cassette file system and the baud rate to be used. On entry X contains a number to set the baud rate.

X=0   Default rate (1200 baud)
X=3   300 baud
X=12   1200 baud

OSBYTE call with A=&90 (144) TV

This call is functionally equivalent to `*TV`. The contents of the X register are used to control the vertical position of the screen display. For example, setting X=2 would move the display two text lines up the screen. Setting X=253 would move the display three lines down the screen. The contents of Y should be either 0 or 1 on entry. Y=0 gives an interlaced display and Y=1 gives a non-interlaced display. Note that the offset and interlace mode selected only come into effect at the next `MODE` change. The values set stay in force until a hard reset. Interlace is off on power up.

*TV is used as follows.

*TV on its own is equivalent to ***TV0,0** and will turn the interlace on at the next **MODE** change or **BREAK**.

*TVx will also turn the interlace on while giving a vertical offset of x. To maintain interlace off when giving an offset you must use ***TV**x**,1** (interlace on may make the picture flicker on some TVs or monitors).

OSBYTE call with A=&91 (145) Get character from buffer

This enables characters to be removed from various input buffers. On entry X indicates the buffer from which the character is to be extracted. On exit Y contains the character and C=0 if a character was successfully removed. If the buffer was empty then C will be 1. The buffer numbers are as follows:

X=0   Keyboard buffer
X=1   RS423 input buffer

OSBYTE calls with A=&92 to &97 Read or write to memory mapped input/output

This group of calls is used to read or write data from or to the various memory mapped input/output devices. It is vital that users use these routines rather than attempting to address devices directly. The use of these routines will ensure that programs will work whether they are executed in the input/output processor (the BBC Microcomputer) or in a second processor. If the user insists on addressing I/O ports directly (eg **STA &FE60**) then he or she will have to rewrite programs when the system is expanded. Considerable effort has been expended to ensure that suitable routines are provided to enable the Assembly Language programmer to expand his or her system painlessly. Please learn to use the facilities provided!

There are three memory mapped input/output areas and these are named FRED, JIM and SHEILA. SHEILA contains all the machine's internal memory mapped devices, such as the analogue to digital converter, and should be treated with considerable respect. FRED and JIM, on the other hand, address external units connected to the 1MHz expansion bus.

Name	Memory address range	OSBYTE call	
		Read	Write
FRED	&FC00-&FCFF	&92(146)	&93(147)
JIM	&FD00-&FDFF	&94(148)	&95(149)
SHEILA	&FE00-&FEFF	&96(150)	&97(151)

On entry to the routines A contains the OSBYTE call number and X the offset within the page. If a byte is to be written it should be in Y.

An application note entitled 'BBC Microcomputer application note no. 1 – 1MHz bus' explains suggested memory allocations for FRED and JIM. It can be purchased from Acorn Computers. SHEILA addresses the devices with the offsets shown in the table.

The user should be aware that the computer expects to service interrupts from all except the user port on the B side of the 6522. The A side is used for the parallel printer interface. Routines are provided (such as OSBYTE and OSWORD calls, etc) to handle these devices. The only circuit that the user should need to handle directly is the 6522 user port. Information about other ports is given for information only – not to encourage you to access the circuits directly. You would be well advised to use the numerous routines provided wherever possible. Further details about some ports are given in the section in chapter 45 which deals with input/output.

SHEILA addresses (offset from &FE00)

Hex offset	Integrated circuit	Register destination	Description Write	Read
00	6845 CRTC		Address register	
01	6845 CRTC		Register file	
08	6850 ACIA		Control register	Status register
09	6850 ACIA		Transmit data register	Receive data register
10	Serial ULA		Control register	
20	Video ULA			
21	Video ULA			
30	LS161		Paged ROM/RAM ID	
34			Shadow RAM select (top bit)	
40	6522 VIA		MOS input/output	
60	6522 VIA	ORB/IRB	Output register 'B'	Input register 'B'
61	6522 VIA	ORA/IRA	Output register 'A'	Input register 'A'
62	6522 VIA	DDRB	Data direction register 'B'	
63	6522 VIA	DDRA	Data direction register 'A'	
	6522 VIA	T1C-L	T1 low-order latches	T1 low-order counter
65	6522 VIA	T1C-H	T1 high-order counter	
66	6522 VIA	T1L-L	T1 low-order latches	
67	6522 VIA	T1L-H	T1 high-order latches	
68	6522 VIA	T2C-L	T2 low-order latches	T2 low-order counter
69	6522 VIA	T2C-H	T2 high-order counter	
6A	6522 VIA	SR	Shift register	
6B	6522 VIA	ACR	Auxiliary control register	
6C	6522 VIA	PCR	Peripheral control register	
6D	6522 VIA	IFR	Interrupt flag register	
6E	6522 VIA	IER	Interrupt enable register	
6F	6522 VIA	ORA/IRA	Same as register 1 except no 'handshake'	
80	8271 FDC		Command register	Status register
81	8271 FDC		Parameter register	Result register
82	8271 FDC		Reset register	
83	8271 FDC		Illegal	Illegal
84	8271 FDC		Write data	Read data
80	1770 FDC		Drive select	
84	1770 FDC		Control	Status
85	1770 FDC		Track	Track
86	1770 FDC		Sector	Sector
87	1770 FDC		Data	Data
A0	68B54 ADLC	CR1/SR1	Control register 1	Status register 1
A1	68B54 ADLC	CR2/SR2	Control register 2/3	Status register 2
A2	68B54 ADLC	TxFIFO/ RxFIFO	Transmit FIFO, continue	Receive FIFO
A3	68B54 ADLC	TxFIFO/ RxFIFO	Transmit FIFO, terminate	Receive FIFO
C0	µPD7002		Data latch, A/D start	Status
C1	ADC			High byte of result
C2				

A few examples will help to clarify the use of these calls.

```
*FX147,5,6
```

would write to FRED+5 (&FC05) the value 6. Similarly

```
LDA #&97
LDX #&62
LDY #&FF
JSR &FFF4
```

would write &FF into location &FE62. An OSBYTE call with A=&97 will write to SHEILA. The base address of SHEILA is &FE00 and to this is added the offset in X (&62). The value written is contained in Y. The net effect is to write to the 6522 data direction register and to cause all the PB lines to become outputs.

OSBYTE call with A=&98 (152) Examine specified buffer

This call examines a buffer. The buffer number (same as for OSBYTE calls 21 and 138) must be in X, and the call returns as follows:

– Carry bit set if buffer empty.
– Carry bit clear if characters present in buffer.
– Y contains the next character which will be returned if the buffer is read.

(Note that the character in Y has not actually been removed from the buffer.)

OSBYTE call with A=&9E (158) Read from speech processor

See Speech System User Guide.

OSBYTE call with A=&9F (159) Write to speech processor

See Speech System User Guide.

OSBYTE call with A=&D1 (209) Enable/disable speech

This call reads (or writes to) location &261, which contains the value sent to the speech processor when speech is output. A value of &50 is the default value, representing speech on. Writing to the location with X=1 will turn speech off. To write to &261, set X=new value, Y=0. To read the location, set X=0, Y=255. On exit, X=the contents of location &261 (if reading), or the previous value held (if writing). The contents of the next location (ie &262) are returned in Y.

OSBYTE with A=&D2 (210) enable/disable sound

This call reads (or writes to) location &262, which contains the sound status. A value of zero is the default value, representing sound on. Writing to this location with X=1 will turn sound off. To write to &262, set X=new value,

Y=0. To read the location, set X=0, Y=255. On exit, X=the contents of location &262 (if reading), or the previous value held (if writing). The contents of the next location (ie &263) are returned in Y.

OSBYTE with A =&DA (218) Read/write number of items in VDU queue

This call reads (or writes) the number of VDU parameters which are still expected. To read the location containing the value of this parameter (&26A), set X=0, Y=&FF. To write to the location, set X=new value, Y=0. Setting X=0, Y=0 gives a useful way of abandoning a VDU queue, otherwise writing to &26A is not recommended. On exit, X contains the 2's complement negative number of bytes still required for the execution of a VDU command. The contents of next location (ie &26B) are returned in Y.

OSBYTE call with A=&E0 (224) Cancel VDU queue

Many VDU codes expect a sequence of bytes. For example, `VDU19`should be followed by five bytes. This call signals the VDU software to throw away the bytes that it has received so far. On entry X and Y must contain zero.

The next group of OSBYTE calls (&E1 to &E8) can be used to read or write status information. The calls read the current value of the status being investigated, AND the value with the contents of Y, EOR the result with the contents of X and then write the value back. If V represents the particular status in the computer then V becomes (V AND Y) EOR X. This sequence enables V to be read or be written to and enables any single bit, or group of bits of V, to be set, cleared or inverted.

If, on entry, X=&00 and Y=&FF the net effect will be to read the value of V into X without altering V. On the other hand if Y=&00 then the contents of X will be written into V. To set a single bit of V without altering other bits set all the bits of Y to 1 except the specified bit. Clear all the bits of X to 0 except the specified bit. For example, to set bit 0 of V to 1 use Y=&FE and X=&01.

To clear a single bit in V set all the bits of Y to 1 except the specified bit and set X=0. For example, to clear bit 0 of V use Y=&FE and X=&00. Of course, many bits may be set, cleared, inverted or examined at the same time.

OSBYTE call with A=&E1 (225) Set base number for function key codes

Normally the red function keys can be programmed to produce strings of characters by, for example, the statement

```
*KEY 0 PRINT
```

As an alternative the keys can produce a single ASCII code. The statement

`*FX225,240`

would set the base value for the function keys to 240 thus causing key **f0** to produce ASCII code 240, **f1** to produce 241 and so on to **f9** which would produce 249. This enables these keys to produce ASCII codes for user defined characters.

`*FX225,1` returns the keys to their normal function of generating strings.

`*FX225,0` makes the keys have no effect.

On entry Y must contain zero.

OSBYTE call with A=&E2 (226) Set base number for SHIFT function key codes

Pressing one of the function keys while the **SHIFT** key is pressed will normally produce ASCII codes in the range 128 to 137. These values were chosen with the Teletext codes in mind.

Shift function key	ASCII code	Teletext effect
f0	128	Nothing
f1	129	Red alphanumeric
f2	130	Green alphanumeric
f3	131	Yellow alphanumeric
f4	132	Blue alphanumeric
f5	133	Magenta alphanumeric
f6	134	Cyan alphanumeric
f7	135	White alphanumeric
f8	136	Flash
f9	137	Steady

These codes are said to have a 'base value' of 128 since key **f0** produces a code 128.

If the user wishes, the base value of the ASCII codes can be changed by using this call:

`*FX226,144`

This would set the **SHIFT** function key codes to produce equivalent graphics color Teletext codes. The default setting is

`*FX226,128`

On entry Y must contain zero.

OSBYTE call with A=E3 (227) Set base number for CTRL function key codes

See the entry for OSBYTE &E2. The default base value is 144.

OSBYTE call with A=&E4 (228) Set base number for SHIFT CTRL function key codes

See entry for OSBYTE &E2. The default is that these key combinations have no effect. *Note:* Remember that pressing **CTRL** and **SHIFT** together stops screen output while they are pressed.

OSBYTE call with A=&E5 (229) ESCAPE key gives ASCII code

This call can be used to make the **ESCAPE** key generate an ASCII code (27 or &1B) instead of interrupting a BASIC program. If X=0 then the **ESCAPE** key interrupts the BASIC program (***FX229,0**). On the other hand, ***FX229,1**(X=1) causes the key to generate its ASCII code. On entry Y must contain zero.

OSBYTE call with A=&E6 (230) Enable/disable normal ESCAPE key action

When a BASIC program is interrupted by pressing the **ESCAPE** key, or by ***FX125**, all internal buffers will be cleared. This call can be used to stop the flushing of all internal buffers when a program is stopped. On entry Y must contain zero.

*FX230,0will permit all buffers to be flushed.

*FX230,1will ensure that no buffers are flushed.

OSBYTE call with A=&E7 (231) Enable/disable user 6522 IRQ

This call sets a 'mask' byte which the operating system uses when servicing IRQs which may have originated in the 6522 which is used for the parallel printer and user port. The operating system ANDs the mask byte with the 6522 interrupt flag register AND the 6522 interrupt enable register. Setting the mask to zero would prevent the operating system from handling the 6522 interrupts thus leaving them available to be handled by user supplied routines. Additionally sideways ROMs may handle (via the operating system) interrupts generated from the B side of the 6522. The mask byte could hide those interrupts from the sideways ROMS. The mask byte defaults to &FF.

OSBYTE call with A=&E8 (232) Enable/disable 6850 ACIA IRQ

This call sets a 'mask' byte which is used by the operating system when servicing IRQs which may have originated from the 6850 ACIA used for the RS423 and cassette interfaces. The operating system ANDs the mask byte with the 6850 status register. See the entry above (OSBYTE &E7) for further comments. The default value is &FF.

OSBYTE call with A=&E9 (233) Enable/disable system 6522 IRQ

As for &E7 but affects the system 6522. The system 6522 is used extensively in the normal operation of the machine and consequently this call should be used with extreme care.

OSBYTE call with A=&EB (235) Presence of speech processor

This call returns presence of speech processor.
X=&FF if speech processor present
X=0 if speech processor not present

OSBYTE with A=&EF (239) Read/write shadow mode state

This call reads (or writes to) location &27F, which contains the shadow mode flag. To read the flag, set X=0, Y=255. To write to &27F, set X=new value, Y=0. On exit, X=0 if in shadow mode, X=1 otherwise (reading). If writing, X returns the previous value of the flag. The contents of next location (ie &26C) are returned in Y. (See chapter 42 for details of the shadow screen facility.)

OSBYTE call with A=&FD (253) Last reset type

This call returns a number indicating what sort of reset last occurred.

Y = 0 Soft break
Y = 1 Power-on break
Y = 2 **CTRL BREAK**

44 An introduction to assembly language

Machine code and the assembler

The heart of any computer, the part that actually does all the processing, is the *central processor unit* (CPU). It is important to realise that no matter what language is typed in at a computer's keyboard (sometimes called the *source language*), the only language that the CPU understands is *machine code*. Of course, the exact form of the machine code depends upon the source language and also upon the type of CPU, but generally speaking any machine code instruction would look something like this:

1010100110000001

This hardly looks like an intelligible instruction, and even rewriting it as two pairs of hexadecimal digits hardly makes it look any better:

A9 81

However, to the BBC Microcomputer's 6512 microprocessor, A9 means 'load the accumulator', and the whole of the above instruction means 'load the accumulator with hexadecimal 81'. (The accumulator is one of six registers which reside in the 6512 microprocessor.) The 'A9' part is known as the 'operation code' or opcode for short, and the 81 is the 'operand'. A few more hex machine code instructions with their meanings are:

A2 64   'load the X register with &64'
A0 00   'load the Y register with zero'
20 F4 FF   'jump to the subroutine at location &FFF4'

You will probably agree that whilst it would be possible to write programs using opcodes like those shown above, it would be extremely tedious. What's more, since the 6512 only carries out one instruction at a time, a sequence of instructions like that shown above would have to be 'poked' into contiguous locations in RAM. As a final indignity, if you spent a long time writing a machine code program which turned out not to work, it would be extremely difficult to trace the errors – unlike BASIC, machine code does not generate error messages automatically.

Fortunately, the BBC Microcomputer contains a program which will generate machine code instructions for you. It's available as part of the BBC BASIC language, and is known as the assembler. The language that the assembler understands is simply known as *assembly language*, or 6502 assembly

language in the BBC Microcomputer's case. (The 6512 is a close relative of the 6502, and uses the same assembly language.)

The assembly language statements which would generate the machine code statements shown above are:

`LDA #129`	(=A9 81,	'load the accumulator with &81)
`LDX #100`	(=A2 64,	'load the X register with &64')
`LDY #0`	(=A0 00,	'load the Y register with zero')
`JSR &FFF4`	(=20 F4 FF,	'jump to the subroutine at location &FFF4')

Clearly, an instruction like 'LDX' looks a lot more like 'load the X register' than 'A2' does. LDA, LDX, LDY and JSR are all known as 'assembly code mnemonics'. 6502 assembly language has 56 mnemonics, some of which will be discussed in detail later in this chapter.

Uses of assembly language

BASIC is a very easy language to use; most of its statements look much like English, and it is very 'friendly', since it gives you helpful messages about mistakes that you may have made in your program. The price to be paid for this 'friendliness' is in memory usage and speed of execution. Assembly language is used where high speed is vital (such as in an arcade game) or where the minimum amount of memory must be used. BASIC is used where ease of programming is more important than either speed or memory usage.

The main features of 6502 assembly language

It is not the purpose of this chapter to teach you how to program in assembly language; you are referred to the large number of books that have been published which are concerned with the BBC Microcomputer and how to program it. This chapter does, however, describe the most important features of assembly language. Below is an example of a simple hybrid BASIC/assembly language program, which you may have already seen in the previous chapter:

```
 10 OSBYTE=&FFF4
 20 P%=&3500
 30 [
 40 LDA #129
 50 LDX #100
 60 LDY #0
 70 JSR OSBYTE
 80 RTS
 90 ]
100 CALL &3500
```

If you have not already seen this program, then what it does is to wait one second for a key to be pressed. If no key is pressed within one second then nothing happens (command mode is returned to). If a key is pressed within one second, its character is printed on the screen and command mode is returned to immediately.

Line 10 simply assigns the value &FFF4 to the variable called OSBYTE. If you are familiar with operating system calls then you will recognise OSBYTE as the name of a MOS rotine and &FFF4 as its call address. Line 20 assigns the value &3500 (an address, as it happens) to the resident integer variable P%. When the program is run, the value in P% is transferred by the machine operating system (MOS) into a register in the 6512 microprocessor called the 'program counter' (abbreviated to PC). The program counter is one of six registers which reside in the 6512. The registers, their abbreviations and their uses are listed below.

The 6512 registers

Program counter – PC (PCL – low byte, PCH – high byte)

A 16-bit register which contains the address of the next instruction to be executed. In the example shown above, the initial value of PC is taken from P%. The contents of the program counter are altered by 'jump' and 'branch' instructions, thereby diverting the flow of the program.

Accumulator – A

An 8-bit general purpose register used for all arithmetic and logical operations.

X register – X

An 8-bit general purpose register often used to contain entry and exit parameter values for MOS routine calls, also used to contain the 'offset' for indexed addressing modes (see later), or as a counter.

Y register – Y

An 8-bit general purpose register often used to contain entry and exit parameter values for MOS routine calls, also used to contain the 'offset' for indexed addressing modes (see later), or as a counter.

Program status register – P

An 8-bit register set up and used by the microprocessor itself. Each bit has its own meaning, concerned with the results of arithmetic and logical operations, and with interrupt status. A detailed description of this register is beyond the scope of this manual. Of occasional interest is the state of bit 0, the carry flag (abbreviated to C). This is set if a carry occurs during an add operation, and is cleared if a borrow occurs during a subtract operation. Following an operating system call, the state of C has a significance which varies according to the particular call involved.

Stack pointer – SP

An 8-bit register which contains the least significant byte of the address of the next free stack location (the most significant byte is always &01). The stack is a portion of memory (&100-&1FF) used for the temporary storage of data (such as return addresses from subroutines). Data is 'pushed' onto the stack in sequence, then removed by 'pulling' it off again. The last byte to be pushed on is the first byte to be pulled off (this is often referred to as a 'last in, first out' queue).

The assembler delimiters '[' and ']', and general assembly language syntax rules

Assembly language statements within a BASIC program must be enclosed between a pair of square brackets (see lines 30 and 90 in the above example). When the BASIC program is RUN, the assembly language statements between the square brackets are assembed into machine code, which is inserted into memory starting at the address specified by P%. An assembly language program consists of a number of assembly language statements, separated by new lines or colons (as in BASIC).

Each assembly language statement consists of an optional label (which must always be preceded by a dot), followed by an instruction. An instruction consists of a three letter assembly language mnemonic followed by an operand (or an address) (both of the latter may be absent depending upon the mnemonic). If a label is included, it must be separated from the mnemonic by at least one space. There need not be any spaces between the mnemonic and the operand. Any character following the operand and separated by at least one space from it will be ignored by the assembler which will move on to the next colon or line for the next statement. A comment may be placed after the operand and should be preceded by a backslash (\). Any text following a backslash in an assembly language statement will be ignored by the assembler up to the next colon or end-of-line.

Line 40 of the example program could therefore be re-written as:

```
.start LDA #129 \load accumulator with OSBYTE number
```

or as

```
.start LDA#129\load accumulator with OSBYTE number
```

(Here, the label `.start` would have no fuction, but would not affect the processing.)

Addressing modes

Most assembly language instructions require data to work on, which must be provided in the operand field of the assembly language statement. Often, this data is an address. The assembler allows several different methods of providing these addresses or data to be used, these methods being known as *addressing modes*. Not all assembly language instructions can use all the addressing modes; see the table in the appendix for more details.

Implicit addressing

This is the simplest form of addressing, which does not require an address to be supplied at all; the address is implied by the instruction itself. For example an `RTS` instruction (see line 80 above) always causes the processor to jump to the bottom two bytes of the stack, which contain the return address to the main program.

Immediate addressing and zero page addressing

Line 40 of the example is

```
LDA #129
```

which means 'load the accumulator with decimal 129' (hexadecimal 29 could be loaded by the statement `LDA #&29`). Here the statement uses the data supplied in the operand field without having to look for it in memory, hence the name 'immediate addressing'. The data can be supplied as a variable, hence

```
LDX #value
```

would load the X register with the value of the variable `value`.

The role of the `#` character in the above examples is important, since it indicates to the assembler that immediate addressing is to be employed, using the data supplied immediately to the right of the `#`. The instruction

```
LDA 129
```

means something quite different; it means 'load the accumulator with the contents of memory location 129'. The computer's main memory is divided into

256 'pages' each of 256 bytes. Page 0 extends from location 0 to location 255 (ie any address with the two most significant bytes set to zero), hence the addressing mode exemplified above is known as 'zero page addressing'. The assembler will automatically select zero page addressing mode (if appropriate) when the address supplied is less than &100 (decimal 256).

Absolute addressing

This addressing mode is very similar to zero page addressing, except that any address in memory can be specified. For example,

```
LDA &3456
```

would load the accumulator with the contents of memory location &3456,

```
LDA 8200
```

would load the accumulator with the contents of memory location 8200.

Indirect addressing

This addressing mode uses an address which is stored in memory. Only two assembly language mnemonics use this mode, `ADC` and `JMP`. For example,

```
JMP (&2010)
```

means 'jump to the location whose address is held in &2010 (least significant byte) and &2011 (most significant byte)'. Note that indirect addressing is indicated to the assembler by enclosing the address in brackets.

Indexed addressing

This addressing mode exists in several forms, all of which share the common feature that two addresses are given: a base address and an offset, or index. An example is shown below:

```
LDA &1F00,X
```

This means 'load the accumulator from the value held at &1F00+X'. X is used here as the index register; Y could be used in the same way. This form of indexed addressing is known as absolute indexed addressing. Note that X (or Y) must be in the range 0-255. Zero page, X indexed addressing is similar except that the base address must be in page zero. For example,

```
LDY &74,X
```

means 'load the Y register with the contents of (&74+X)'

The assembler automatically uses this mode, if available, if a page zero address is specified in the operand field. Note that the offset must be supplied in the X register, except for the `LDX` mnemonic when the Y register can (and must) be used. Note that the sum of the base address plus the offset in this mode will

always be taken as an address in page zero. If the address moves out of page zero the processor will perform a 'wrap-around' operation to take it back into page zero – for example an address of &102 would be wrapped around to &002.

Another type of indexed addressing is pre-indexed indirect addresing. An example of the instruction format is:

```
LDA (&82,X)
```

This instruction adds the address in the X register to &82 to give a new address; the contents of the location at this new address, and the contents of the location above it, together supply the full 16-bit address from which the accumulator is loaded. This addressing mode is designed for use with a table of addresses in zero page locations. For example, if we have

```
?&70=&00
?&71=&20
?&72=&FF
?&73=&21
```

then

```
LDX #0          \ set X to zero
LDA (&70,X)     \ A=?&2000, ie address in (&70+X),
(&71+X)
```

<perform some other operation>

```
LDX #2          \ set X to 2
LDA (&70,X)     \ A=?&21FF, ie address in (&72),
(&73)
```

Note that the base address, and the base address plus the offset must be in page zero (the wrap around operation described above still applies). The Y register cannot be used for this addressing mode. The mode is called pre-indexed because the index is first added to the base address to give the address of the pair of locations which hold the address loaded into the accumulator. In post-indexed indirect addressing, the 16-bit address held in the location pair given by the base address is extracted first, and the index is then added to this address to give the address of the location from which the accumulator is loaded. An example of the instruction format is:

```
LDA (&80),Y
```

Note that in this addressing mode the Y register is used as the index (X cannot be). The 'zero page' and 'wraparound' comments given above still apply. An example of its use is shown below. The program shown sets 128 memory locations to &FF, starting at the address contained in locations &80 (low byte) and &81 (high byte):

```
10 ?&80=&00:?&81=&28
 20 DIM GAP 50
 30 P%=GAP
 40 [
 50       LDY #0       \ set loop index to zero
 60       LDA #&FF      \ set value to be loaded
 70 .loop STA (&80),Y  \ ?(2800+Y)=&FF, base addr.
in &80 and &81
 80       INY           \ Y=Y+1
 90       CPY #128      \ end of loop reached?
100       BNE loop      \ if not, go to loop
110       RTS
120 ]
130 CALL GAP
```

(Lines 20, 30 and 130 are explained below.)

Relative addressing

Relative addressing is the addressing mode used by assembly language branch instructions. In the above example, the mnemonic at line 100 is such an instruction. If the condition tested at line 100 is not satisfied, ie if Y=128, the next instruction to be executed will be the one being pointed to by the program counter – the instruction at line 110. If however the condition is satisfied, ie if Y<>128, the processor will decrement the program counter so that it points to the insruction labelled by **loop**, and this instruction will be the next one to be executed. In the above case, the program counter has to be decremented by 7 (this will become apparent from a careful study of the machine code printed on the screen when the above program is **RUN**). The address of the instruction labelled by **loop** is -7, relative to the address held in the program counter. This relative addressing is carried out by the assembler, and normally goes unnoticed by the programmer. There are, however, limits to the number of bytes which can be jumped forward or back; a branch of up to 127 bytes forward or 128 bytes back from the Program counter value (at the time the branch instruction is being executed) is allowed. Care should therefore be taken that labels are not too far from the instructions that branch to them, otherwise an 'Out of range' error will result. (Remember that each instruction may be 1, 2 or 3 bytes long.)

Accumulator addressing

Th final addressing mode used by the assembler is known as accumulator addressing. This is where the accumulator is addressed rather than a memory location, and is specified by placing A in the operand field. For example:

ASL A

means 'shift the contents of the accumuator one bit to the left'. Note that this means that 'A' cannot be used as a variable name within an assembly language program.

Finally, it must be re-emphasised that each assembly language mnemonic can only use some of the addressing modes detailed above; see the table in the appendix for the addressing modes used by each instruction.

Placing machine code programs in memory

When the assembler is creating a machine code program the code produced is placed in memory starting from the address in P% (unless O% is used, see **OPT** below). The assembler updates the value in P% as it is assembling, and at the end of an assembly operation the value in P% represents the address of the first 'free' memory location after the machine code program. In the example program shown above, the value of P% is allocated directly (at line 20). This method of setting up P% is somewhat dangerous, since you have to be sure that location &3500 and (in this case) the nine locations beyond &3500 do not contain anything important. A much safer way of setting up P% is to dimension a block of memory using a variation of the BASIC **DIM** keyword. The example program below illustrates this feature, and other features used with assembly language:

```
  5 REM Uses assembly code to change to mode 4 and
draw a triangle
 10 OSWRCH=&FFE3
 20 DIM GAP% 100
 30 DIM data &1C
 40 FOR opt%=0 TO 3 STEP 3
 50 P%=GAP%
 60 [
 70 OPT opt%
 80 .entry LDX#0      \ set data block offset to
zero
 90 .loop  LDA data,X \ load VDU parameter from
data block
100        JSR OSWRCH \ perform VDU command
110        INX        \ increment offset
120        CPX #&1C   \ has all data been loaded?
```

```
130        BNE loop   \ if not, load next item
140        RTS        \ return to BASIC
150 ]
160 NEXT opt%
170 !data=&04190416
180 data!4=&00C800C8
190 data!8=&00C80119
200 data!&C=&01190000
210 data!&10=&00ADFF9C
220 data!&14=&FF9C0119
230 data!&18=&0000FF53
240 CALL entry
```

This program uses OSWRCH to perform operations equivalent to BASIC's **VDU** command, drawing a triangle on the screen in **MODE 4**. Note the use of indexed addressing at line 90 to load values from a table. Extensive use is made of indirection operators (see chapter 39 for details). The first pass of the assembly language loop is equivalent to

VDU &16,&04 (see line 170, reading from the right)
or **VDU 22,4** (ie change to **MODE 4**).

Note line 20. This use of the **DIM** statement causes (in this case) 100 bytes of memory to be reserved, the start address of the block being transferred to the variable **GAP%**. The start address will always be greater than the value of TOP and below the start of screen memory, and so is 'safe' (the allocation is made in the same way as the allocation of a BASIC variable). The value in **GAP%** is transferred to P% at line 50, and so gives the address where the machine code will be assembled. Line 30 uses **DIM** to reserve 28 memory locations for the data table to be used by the program, the start address of the table being stored in the variable data.

Although for most applications machine code will be assembled and run at the same place in memory, it is possible to assemble code at one location and run it at another. Setting the resident integer variable O% to an address (and at the same time setting the pseudo variable **OPT** to a certain value, see below) will cause the machine code to be assembled at that address but assembled to run at the address given by P%; in other words the program counter is still controlled by P% during assembly. O% can be used, for example, to get a machine code program to run in the I/O processor which has been assembled in a second processor. See the description of the **OPT** keyword below for further details.

OPT, forward referencing and two-pass assembly

Take a look at the following program:

```
 10 DIM GAP 50
 20 P%=GAP
 30 address=&70
 40 [
 50 LDA address    \ Load accumulator with
    contents of address
 60 CMP #0          \ Compare accumulator
    contents with zero
 70 BEQ zero        \ Jump to label if
    accumulator contents zero
 80 STA address+1 \ Store accumulator
    contents at address+1
 90 .zero RTS
100 ]
```

In this program, line 70 is known as a 'forward reference', since it refers to a label which doesn't appear until line 90. If the above program is run, the error message

```
No such variable at line 70
```

will appear. This is because the assembler has not allocated an address to the label **zero** yet. The reason for the error message is that a 'two-pass assembly' is required. The first pass allocates addresses to all labels; the second pass can then jump to the correct memory location when a 'jump to label' instruction is encountered. A two-pass assembly is controlled using the **OPT** command, as shown below:

```
 10 DIM GAP 50
 20 address=&70
 30 FOR pass=0 TO 3 STEP 3
 35 P%=GAP
 40 [
 45 OPT pass
 50 LDA address    \ Load accumulator with
    contents of address
 60 CMP #0          \ Compare accumulator
    contents with zero
 70 BEQ zero        \ Jump to label if
    accumulator contents zero
 80 STA address+1 \ Store accumulator
    contents at address+1
 90 .zero RTS
```

```
100 ]
110 NEXT
120 CALL GAP
```

Here, the assembly language is enclosed in a loop such that two passes are made, with the value of **OPT** as 0 on the first pass and 3 on the second pass. (The same device is employed in the triangle drawing example above.) The values which are assigned to **OPT** have the following effects:

0	Assembler errors suppressed, no listing
1	Assembler errors suppressed, listing
2	Assembler errors reported, no listing
3	Assembler errors reported, listing

So, in the above example, **OPT=0** on the first pass so there will be no listing and no errors reported. This allows the forward referenced label to be identified without the assembly being interrupted. On the second pass, **OPT=3** and so a listing of the compiled code is produced, along with any programming errors. Note that the assignment statement **P%=GAP** must be enclosed within the loop so that it is reset before each pass. If the above program is **RUN**, its assembly and execution should now be successful. (The program is trivial; it merely loads a byte from the memory location specified by address, and, if the byte is non-zero, transfers it to the next memory location).

Setting **OPT** equal to 4, 5, 6 or 7 has the same effect as setting it to 0, 1, 2 or 3 (respectively) except that the code is placed in memory starting at the address supplied in O%, rather than that in P%.

Note that **OPT** is not an assembly language mnemonic, but a so-called 'pseudo-operation', or assembler directive. It tells the assembler to do something, but is not an assembled instruction.

The EQUate facility

EQU is a pseudo-operation, which allows data to be incorporated into the body of an assembly language program. The **EQU** operations available are:

EQUB   equate byte   reserves one byte of memory

EQUW   equate word   reserves two bytes of memory

EQUD   equate double word   reserves four bytes of memory

EQUS   equate string   reserves memory as required

These operations set the reserved memory locations to the values specified in the operand field. The operand field may contain a string (in double quotes) or a string variable for the **EQUS** operation, or a number or a numeric variable for the other **EQU** operations. The assembler will use the least significant part of the value if too large a value is specified. As an example of the use of **EQUD**, lines 30 and 170 to 230 of the triangle drawing example could be replaced with:

```
141 .data EQUD &04190416
142      EQUD &00C800C8
143      EQUD &00C80119
144      EQUD &01190000
145      EQUD &00ADFF9C
146      EQUD &FF9C0119
147      EQUD &0000FF53
```

The following example program illustrates the effects of including each of the **EQU** pseudo-operations within an assembly language program:

```
10 P%=&3000
20 A$="stringvar"
30 [
40 EQUS "string"
50 EQUS A$
60 EQUB 180
70 EQUW 12500
80 EQUD 6E6
90 ]
>RUN
3000
3000 73 74 72
     69 6E 67 EQUS "string"
3006 73 74 72
     69 6E 67
     76 61 72 EQUS A$
300F B4       EQUB 180
3010 D4 30    EQUW 12500
3012 80 8D 5B
     00       EQUD 6E6
```

The above printout shows that location &3000 contains &73, the hexadecimal ASCII code for 's', &3001 contains the hexadecimal ASCII code for 't', etc

Machine code entry points

The BBC Microcomputer is unusual in a number of respects, not least because of the care taken to ensure that everything that can be done by programs written in the input/output processor (the BBC Microcomputer) can also be done in the second processor which is on the far side of the Tube .

If a piece of machine code alters a particular memory location that controls the screen display directly, then that same piece of machine code will not work in the second processor because the screen will not be affected by any memory location in the second processor.

It is vital that programmers avoid reading and writing to specific memory locations such as the screen memory, zero page locations used by BASIC, and memory mapped input/output devices. System calls are provided to enable you to access all these important locations and use of these system calls will ensure that your programs interact successfully with the machine. Don't feel that we are trying to hide anything from you – on the contrary we are offering you access to all the I/O routines that BASIC uses! Cultivate the habit of using system calls and then you will not need to rewrite your code when you move it to the second processor.

45 The operating system calls

Machine code user programs should communicate with the operating system by calling routines in the address range &FF00 to &FFFF. These routines then call a specific internal routine whose address may change in different operating systems. The address of the specific routine is held in RAM between locations &200 and &2FF. The user may change the address held in these RAM locations to intercept any operating system call he or she wishes.

Thus the 'output the character in A' routine is entered at &FFEE in all environments. The routine indirects through location &20E in all machines. The contents of locations &20E and &20F will vary depending on the machine and the version of the operating system. In one particular machine the address in &20E and &20F is &E0A4 which is the local internal address of the normal 'output the character in A' routine.

Parameters are passed to the routines in various ways using either the 6512 A, X and Y registers, zero page locations or a parameter block. All routines should be called with a `JSR` and with the decimal flag clear (ie in binary mode).

In the detailed descriptions which follow A refers to the accumulator; X and Y refer to the registers; C, D, N, V and Z refer to the processor flags.

The table on the next page gives a summary of operating system calls and indirect vectors.

Files

Files are treated as a sequence of eight bit bytes. They can be accessed in one operation (using OSFILE) or in blocks (using OSGBPB) or a byte at a time (using OSBGET and OSBPUT). The following attributes may be associated with each file.
Load address is the address in memory to which the file should normally be loaded. This can be over-ridden when the file is loaded, if necessary.
Execution address is meaningful only if the file contains executable machine code, in which case it is the address where execution should start. If the file contains a high level language program then the execution address is unimportant.
Length is the total number of bytes in the file. It may be zero.
Pointer is an index pointing to the next byte of data that is to be processed. The value of 'pointer' may be read or written (using OSARGS), and it does not indicate whether the appropriate byte has yet been transferred from file to memory or vice versa. Pointer is automatically updated by OSBGET and OSBPUT.

OSWRSC

Writes a byte (contained in A) to the screen. The display memory location to be written should be set up in &D6 (LSB) and &D7 (MSB). Y (on entry) is used to contain an offset from this address. The effect of this call can be illustrated (somewhat crudely!) by the program shown below:

```
10 MODE 7
20 ?&D7=&7C
30 Y%=255:A%=&45
40 FOR J=0 TO 127 STEP 2
50 ?&D6=J
60 CALL(&FFB3)
70 NEXT J
```

On exit, A, X and Y are preserved, C is undefined.

OSRDSC

Reads a byte from the screen, the display memory location concerned being contained in &F6 (LSB) and &F7 (MSB). The byte is returned in A. The following program illustrates the effect of this call:

```
 10 MODE7
 20 FOR K=1 TO 255
 30 PRINT "E";
 40 NEXT K
 50 ?&F7=&7C
 60 VDU14
 70 FOR J=0 TO 127
 80 ?&F6=J
 90 PRINT ~USR(&FFB9)
100 NEXT J
```

This routine can also be used to read bytes from paged ROM, with Y (on entry) set equal to the ROM socket number. In this context, this call has in the past been referred to as OSRDRM.

OSFIND

Opens a file for writing or reading and writing. The routine is entered at &FFCE and indirects via &21C. The value in A determines the type of operation.

A=0	Causes a file or files to be closed.
A=&40	Causes a file to be opened for input (reading).
A=&80	Causes a file to be opened for output (writing).
A=&C0	Causes a file to be opened for input and output (random access).

Routine		Vector		Summary of function
Name	Address	Name	Address	
		UPTV	222	User print routine
		EVNTV	220	Event interrupt
		FSCV	21E	File system control entry
OSWRSC	FFB3	-	-	Write byte to screen
OSRDSC	FFB9	-	-	Read byte from screen
OSFIND	FFCE	FINDV	21C	Open or close a file
OSGBPB	FFD1	GBPBV	21A	Load or save a block of memory to file
OSBPUT	FFD4	BPUTV	218	Save a single byte to file from A
OSBGET	FFD7	BGETV	216	Load a single byte to A from file
OSARGS	FFDA	ARGSV	214	Load or save data about a file
OSFILE	FFDD	FILEV	212	Load or save a complete file
OSRDCH	FFE0	RDCHV	210	Read character (from keyboard) to A
OSASCI	FFE3	-	-	Write a character (to screen) from A plus LF if (A)=&0D
OSNEWL	FFE7	-	-	Write LF,CR (&0A,&0D) to screen
OSWRCH	FFEE	WRCHV	20E	Write character (to screen) from A
OSWORD	FFF1	WORDV	20C	Perform miscellaneous OS operation using control block to pass parameters
OSBYTE	FFF4	BYTEV	20A	Perform miscellaneous OS operation using registers to pass parameters
OSCLI	FFF7	CLIV	208	Interpret the command line given
		/IRQ2V	206	Unrecognised IRQ vector
		/IRQ1V	204	All IRQ vector
		/BRKV	202	Break vector
		USERV	200	Reserved

If A=&40, &80 or &C0 then Y (high byte) and X (low byte) must contain the address of a location in memory which contains the file name terminated with CR(&0D). On exit A will contain the channel number allocated to the file for all future operations. If A=0 then the operating system was unable to open the file.

If A=0 on entry then a file, or all files, will be closed depending on the value of Y. Y=0 will close all files, otherwise the file whose channel number is given in Y will be closed.

On exit C, N, V and Z are undefined and D=0. The interrupt state is preserved, however interrupts may be enabled during the operation.

OSGBPB

The operating system call to get or put a block of bytes to or from a file which has been opened with OSFIND. The routine is entered at &FFD1 and vectors via &21A. This call is not available on the cassette filing system, and is fully documented in the appropriate disc filing system user guides.

OSBPUT

Writes (puts) the byte in A to the file previously opened using OSFIND. The routine is entered at &FFD4 which indirects through &218. On entry Y contains the channel number allocated by OSFIND.

On exit A, X and Y are preserved, N, V and Z are undefined and D=0. The interrupt state is preserved but interrupts may be enabled during the operation.

OSBGET

Gets (reads) a byte from a file into A. The file must have been previously opened using OSFIND and the channel number allocated must be in Y. The routine is entered at &FFD7 which indirects via &216.

On exit C=0 indicates a valid character in A. C=1 indicates an error and A indicates the type of error, A=&FE indicating an end of file condition. X and Y are preserved, N, V and Z are undefined and D=0. The interrupt state is preserved but interrupts may be enabled during the operation.

OSARGS

This routine enables a file's attributes to be read from file or written to file. The routine is entered at &FFDA and indirects via &214. On entry X must point to four locations in zero page and Y contains the channel number.

if Y is non-zero then A will determine the function to be carried out on the file whose channel number is in Y.

A=0   Read sequential pointer.
A=1   Write sequential pointer.
A=2   Read length
A=&FF   Ensure that this file is up to date on the media.

If Y is zero then the contents of A will determine the function to be carried out.

A=0 Return, in A, the type of filing system in use. The value of A on exit has the following significance:

0   No filing system.
1   1200 baud cassette filing system.
2   300 baud cassette filing system.
3   Sideways ROM filing system.
4   Disc filing system.
5   Econet filing system.
6   Teletext filing system
7   IEEE filing system
8   Advanced Disc Filing System

A=1 Return address of the rest of the command line in the zero page locations.

A=&FF Ensure that all open files are up to date on the media.

On exit X and Y are preserved, C, N, V and Z are undefined and D=0. The interrupt state is preserved but interrupts may be enabled during the operation.

OSFILE

This routine, by itself, allows a whole file to be loaded or saved. The routine is entered at &FFDD and indirects via &212.

On entry A indicates the function to be performed. X and Y point to an 18 byte control block anywhere in memory. X contains the low byte of the control block address and Y the high byte. The control block is structured as follows from the base address given by X and Y.

OSFILE control block

00	Address of file name, which must be terminated by &0D	LSB
01		MSB
02	Load address of file	LSB
03		
04		
05		MSB
06	Execution address of file	LSB
07		
08		
09		MSB
0A	Start address of data for write operations,	LSB
0B	or length of file for write operations	
0C		
0D		MSB
0E	End address of data, that is byte after last	LSB
0F	byte to be written or file attributes	
10		
11		MSB

The table below indicates the function performed by OSFILE for each value of A.

A=0   Save a section of memory as a named file. The file's catalogue information is also written.

A=1   Write the catalogue information for the named file (disc only).

A=2   Write the load address (only) for the named file (disc only).

A=3   Write the execution address (only) for the named file (disc only).

A=4   Write the attributes (only) for the named file (disc only).

A=5   Read the named file's catalogue information. Place the file type in A (disc only).

A=6   Delete the named file (disc only).

A=&FF   Load the named file and read the named file's catalogue information.

When loading a file the byte at XY+6 (the LSB of the execution address) determines where the file will be loaded in memory. If it is zero then the file will he loaded to the address given in the control block. If non-zero then the file will be loaded to the address stored with the file when it was created.

The file attributes are stored in four bytes. The least significant eight bits have the following meanings (for the Econet filing system):

Bit Meaning

0   Not readable by you
1   Not writable by you
2   Not executable by you
3   Not deletable by you
4   Not readable by others
5   Not writable by others
6   Not executable by others
7   Not deletable by others

File types are as follows:

0   Nothing found
1   File found
2   Directory found

A BRK will occur in the event of an error and this can be trapped if required. See 'Faults, events and BRK handling' towards the end of this chapter.

On exit X and Y are preserved, C, N, V and Z are undefined and D=0. The interrupt state is preserved but interrupts may be enabled during the operation.

OSRDCH

This routine reads a character from the currently selected input stream into A. The routine is called at location &FFE0 and indirects via &210. The input stream can be selected by an OSBYTE call with A=2. See chapter 43.

On exit C=0 indicates a successful read and the character will be in A. C=1 indicates an error and the error type is returned in A. If C=1 and A=&1B then an escape condition has been detected and the user must at least acknowledge this by performing an OSBYTE call with A=&7E; BASIC will normally do this for you. X and Y are preserved, N, V and Z are undefined and D=0. The interrupt state is preserved.

OSASCI

This routine writes the character in A to the currently selected output stream by using OSWRCH. However, if A contains &0D then OSNEWL is called instead. The actual code at location &FFE3 is

```
FFE3    C9   0D       OSASCI    CMP  #&0D
FFE5    D0   07                 BNE  OSWRCH
FFE7    A9   A0       OSNEWL    LDA  #&0A
FFE9    20   EEFF               JSR  OSWRCH
FFEC    A9   0D                 LDA #&0D
FFEE    6C   0E02     OSWRCH    JMP  (WRCHV)
```

On exit A, X and Y are preserved, C, N, V and Z are undefined and D = 0. The interrupt state is preserved.

OSNEWL

This call issues an LF CR (line feed, carriage return) to the currently selected output stream. The routine is entered at &FFE7.

On exit X and Y are preserved, C, N, V and Z are undefined and D = 0. The interrupt state is preserved.

OSWRCH

This call writes the character in A to the currently selected output stream. The output stream may be changed using an OSBYTE call with A=3. See chapter 43 for more details. OSWRCH is entered at location &FFEE and indirects via &20E.

On exit A, X and Y are preserved, C, N, V and Z are undefined and D=0. The interrupt state is preserved but interrupts may be enabled during the operation.

All character output from BASIC, the operating system and anything else uses this routine. It is, therefore, easy to pass all output to a user provided output routine by placing the address of the user routine at WRCHV (&20E). However, the user should note that all control characters have special significance. For example, &1C is followed by four bytes which define a text window. See chapter 34 on VDU codes for a complete listing of control characters. If the user wishes to intercept any control characters then his or her routine must check for all control characters. The routine must arrange to skip however many bytes follow a particular code since these bytes might, inadvertently, contain a control code. For example, the BASIC statement

```
GCOL 1,3
```

is passed to the operating system as a string of bytes through OSWRCH. In fact, in this case the bytes would be &12,1,3.

OSWORD

The OSWORD routine invokes a number of miscellaneous operations all of which require more parameters or produce more results than can be passed in A, X and Y. As a result, all OSWORD calls use a parameter block somewhere in memory. The exact location of the parameter block is given in X (low byte) and Y (high byte). The contents of A determine the exact nature of the OSWORD call.

All OSWORD calls are entered at location &FFF1 which indirects through &20C. The table below summarises the OSWORD calls.

OSWORD summary

A=	Summary of function
0	Read a line from the current input stream to memory
1	Read the elapsed time clock
2	Write the elapsed time clock
3	Read interval timer
4	Write interval timer
5	Read a byte in the input/output processor memory
6	Write a byte in the input/output processor memory
7	Generate a sound
8	Define an envelope for use with the SOUND statement
9	Read pixel colour at screen position X,Y
A	Read dot pattern of a specific displayable character
B	Read the palette value for a given logical colour

OSWORD with A=0

This routine accepts characters from the current input stream and places them at a specified location in memory. During input the **delete** code (ASCII 127) deletes the last character entered, and **CTRL U** (ASCII 21) deletes the entire line. The routine ends if **RETURN** is entered (ASCII 13) or an ESCAPE condition occurs.

The control block contains five bytes:

00	Address of buffer for input line
01	MSB
02	Maximum length of line
03	Minimum acceptable ASCII value
04	Maximum acceptable ASCII value

Characters will only be entered if they are in the range specified by XY+3 and XY+4.

On exit C=0 indicates that **RETURN** (CR; ASCII code 13 or &D) ended the line. C not equal to zero indicates that an escape condition terminated entry. Y is set to the length of the line, excluding the CR if C=0.

OSWORD call with A=1 Read clock

This call is used to read the internal elapsed time clock into the five bytes pointed to by X and Y. This clock is the one used by BASIC for its **TIME** function. The elapsed time clock is reset to zero when the computer is switched on and if a hard reset is executed. Otherwise it is incremented every hundredth of a second. The only thing that will cause it to lose time is pressing the **BREAK** key and keeping it pressed.

On entry X and Y should point to the memory locations where the result is to be stored. Y contains the high byte and X the low byte of the address.
On exit X and Y are undefined and the time is given in location XY (LSB) to XY+4 (MSB). The time is stored in pure binary.

OSWORD call with A=2 Write clock

This call is used to set the internal elapsed time clock from the five bytes pointed to by XY.

On entry X and Y should point to the memory locations where the new time is stored. Y contains the high byte and X the low byte of the address. The least significant byte of the time is stored at the address pointed to by XY and the most significant byte of the time is stored at address XY+4. A total of five bytes are required.

OSWORD call with A=3 Read interval timer

In addition to the clock there is an interval timer which is incremented every hundredth of a second. The interval is stored in five bytes pointed to by X and Y. See OSWORD with A=1.

OSWORD call with A=4 Write interval timer

On entry X and Y point to five locations which contain the new value to which the clock is to be set. The interval timer increments and may cause an event when it reaches zero. Thus setting the timer to &FFFFFFFFFE would cause an event after two hundredths of a second.

OSWORD call with A=5 Read I/O processor memory

This call enables any program to read a byte in the I/O processor no matter in which processor the program is executing.

On entry X and Y point to a block of memory as follows:

XY	LSB of address to be read
XY+1	
XY+2	
XY+3	MSB of address to be read

On exit the eight bit byte will be stored in XY+4. A further feature is available on machines fitted with OS 2.00. The feature enables an additional 12K of memory to be accessed, which exists as sideways RAM at locations &8000-&AFFF. It is accessed by ROM IDs 128-255 (ie any value with the top bit set), and hence will not receive service calls. Furthermore, the MOS will not find it to contain a language. Bytes may be read from the RAM with the top two bytes of the memory block set to &FFFE. *Note: references to this 12K of memory which are made in this manual are not necessarily applicable to other Acorn products*.

OSWORD call with A=6 Write to I/O processor memory

As pointed out previously, programs that are to work through the Tube must not attempt to access memory locations in the I/O processor directly. This call provides easy access to locations in the BBC Microcomputer wherever the user's program happens to be.

On entry X and Y point to a block of memory initialised as follows:

XY	LSB of address to be changed
XY+1	
XY+2	
XY+3	MSB of address to be changed
XY+4	Byte to be entered at address given

Bytes may be written to the sideways RAM described under OSWORD with A=5 by setting the top two bytes of the memory block to &FFFE. Note that shadow RAM locations &A000 to &AFFF should only be used for user supplied 'VDU driver' machine code programs. When shadow mode is on, the MOS VDU drivers will access shadow display RAM, not 'normal' display RAM (ie &3000 to &7FFF). Any access to &3000 – &7FFF by machine code running in &A000 – &AFFF will automatically be diverted to shadow display RAM. This facility gives faster VDU access when in shadow mode.

OSWORD call with A=7 Make a sound

This call can be used to generate a sound. The eight bytes pointed to by locations XY to XY+7 are treated as four two-byte values. These four values determine the sound effect. See the keyword SOUND for a detailed description.

XY	Channel	LSB	1	01
XY+1		MSB		00
XY+2	Amplitude	LSB	-15	F1
XY+3		MSB		FF
XY+4	Pitch	LSB	200	C8
XY+5		MSB		00
XY+6	Duration	LSB	20	14
XY+7		MSB		00

The example figures on the right of the table show first the required decimal value and secondly the two hexadecimal values required. The figures are only illustrative.

On exit X and Y are undefined.

OSWORD call with A=8 Define an envelope

This call is used to define an envelope which can be used by a **SOUND** statement or equivalent OSWORD call. On entry X and Y point to an address in memory where 14 bytes of data are stored. Y contains the high part of the address and X the low part. The envelope number is stored at XY and the following 13 locations contain data for that envelope. See the entry for the **ENVELOPE** keyword for more details.

On exit X and Y are undefined.

OSWORD call with A=9 Read a pixel

This call enables the machine code programmer to read the status of a graphics point at any specified location. On entry X and Y point to a block of five bytes. Y contains the most significant byte of the address and X the least significant byte. On entry the first four bytes are set up thus:

XY   LSB of X coordinate
XY+1   MSB of X coordinate
XY+2   LSB of Y coordinate
XY+3   MSB of Y coordinate

On exit XY+4 contains the logical colour of the point or &FF if the point is off the screen. X and Y are undefined.

OSWORD call with A=&0A Read character definition

Characters are displayed on the screen as an eight by eight matrix of dots. The pattern of dots for each character in `MODES 0` to `6`, including user defined characters, is stored as eight bytes (see chapter 34). This call enables the eight bytes to be read into a block of memory starting at an address given in X and Y.

On entry the ASCII code of the character is the first entry on the block.

On exit the block contains data as shown below. X and Y are undefined.

XY	Character required
XY+1	Top row of displayed character
XY+2	Second row
.	
.	
.	
XY+8	Bottom row of displayed character

OSWORD call with A=&0B Read palette

The reader will be aware that each logical colour (0 to 15) has an actual (or displayed) colour associated with it. The actual to logical association can be changed with `VDU19`. This OSWORD call enables one to determine the actual colour currently assigned to each logical colour. On entry the X and Y registers contain the address of the start of a block of five bytes. The first byte should contain a value representing the logical colour.

On exit the following four bytes will contain the same four numbers used when `VDU19` assigned an actual colour to the same logical colour. Suppose that logical colour 2 was in fact set to blue (4) by the statement

```
VDU 19,2,4,0,0,0
```

then this call would produce the following result:

XY	2	Logical colour
XY+1	4	Actual colour (blue)
XY+2	0	
XY+3	0 }	Padding zeros for future
XY+4	0 }	expansion

Command line interpreter (&FFF7)

The machine operating system CLI is usually accessed from a high level language by starting a statement with an asterisk. For example:

```
*MOTOR 0,1
```

The command line itself (excluding the asterisk) is then passed, without any further processing, to the CLI.

Machine code programs can use all operating system commands by placing the address of a command line in the X (LSB) and Y (MSB) registers and calling &FFF7. This routine indirects through location &208.

The command line should not start with an asterisk and must end with an &0D. In fact any leading asterisk or spaces will be stripped.

The following BASIC program illustrates this:

```
10 DIM C 20
20 $C="MOTOR 1"
30 X%=C MOD 256
40 Y%=C DIV 256
50 CALL &FFF7
```

When `RUN` the cassette motor will turn on. The computer will have allocated a space for C – perhaps at location &1B0A in which case successive bytes would contain:

Address	Contents	
&1B0A	4D	(M)
&1B0B	4F	(O)
&1B0C	54	(T)
&1B0D	4F	(O)
&1B0E	52	(R)
&1B0F	20	(Space)
&1B10	31	(1)
&1B11	0D	(Return)

Of course, this particular example would have been easier as a `*FX` call or simply as `*MOTOR 1`. However, complex commands may need this call.

Faults, events and BRK handling

It is necessary to provide some means to enable programs to deal with faults such as `Illegal command` or `Division by zero`. BASIC uses the 6502 `BRK` instruction when dealing with faults like this and user written programs can also use the same facility. In BASIC (for example), a `BRK` instruction is followed by a sequence of bytes giving the following information:

– `BRK` instruction, value &00.
– Fault number.
– Fault message (may contain any non-zero character).
– &00 to terminate message.

When the 6512 encounters a `BRK` instruction the operating system places the address following the `BRK` instruction in locations &FD and &FE. Thus these locations point to the 'fault number'. The operating system then indirects via location &202. In other words control is transferred to a routine whose address is given in locations &202 (low byte) and &203 (high byte). The default routine, whose address is given at the location, prints the default message.

The BRK handling outline above enables the user to intercept normal procedures and to generate his or her own special messages and error numbers in user written machine code routines. The `CALL` demonstration program towards the beginning of this chapter shows this in practice. See also IRQ at the end of this chapter.

While faults are in general, 'fatal', there is another class of events, called 'events', which are informative rather than fatal. This class of events includes, for example, a key being pressed on the keyboard. The user may wish to detect such an operation or may be happy to ignore it. When the operating system detects an 'event' then, if that event is enabled (by using `*FX14`) it indirects via &220 with an event code in the accumulator. The contents of X and Y depend on the event. The event codes in A indicate the following:

Accumulator description

0	Buffer empty	X = buffer identity
1	Buffer full	X = buffer identity
		Y = character that could not be stored
2	Keyboard interrupt	
3	ADC conversion complete	
4	Start of TV field pulse (vertical sync)	
5	Interval timer crossing zero	
6	ESCAPE condition detected	

The user supplied event handling routine is entered with interrupts disabled and it should not enable interrupts. The routine should return (`RTS`) after a short period, say one millisecond maximum, and should preserve the processors P, A, X and Y registers.

Interrupt handling

The whole machine runs under continuous interrupts but nonetheless the user can easily add his or her own interrupts and handling routines. Because the machine runs under interrupts, software timing loops should not be used. Several hardware timers are available to the user and these should be used wherever possible.

NMI – non-maskable interrupt

In general these should be avoided. When a disc filing system ROM is fitted NMIs will be handled by the ROM. Again, it should be emphasised that NMI is reserved for the operating system.

IRQ - interrupt request

When an IRQ is detected the operating system immediately indirects through location &204 (IRQ1V) to an operating system routine which handles all anticipated internal IRQS. If the operating system is unable to deal with the IRQ (because it has come from an unexpected device such as the user 6522), then the system indirects through &206 (IRQ2V). Thus the user routine for handling IRQs should normally be indirected via IRQ2V but if top priority is required the user routine can be indirected via IRQ1V.

In either case the user supplied routine must return control to the operating system routine to ensure clean handling.

The operating system handles BRK and IRQs with the following code.

	STA	&FC	\temporary for A
	PLA		
	PHA		\get processor status
	AND	#&10	
	BNE	BRK	
	JMP	(&0204)	\IRQ1V
BRK	TXA		\BRK handling
	PHA		\save X
	TSX		
	LDA	&103,X	\get address low
	CLD		
	SEC		
	SBC	#1	
	STA	&FD	
	LDA	&104,X	\get address high
	SBC	#0	
	STA	FE	

Note that A is stored in location &FC so that it can be accessed by user routines. When the computer indirects through &202 (BRKV), &204 (IRQ1V) and &206 (IRQ2V) X and Y will contain correct values. The user must not enable interrupts during his or her IRQ service routine.

46 Analogue input

The BBC Microcomputer is fitted with a socket at the back marked 'analogue in'. Into this socket you can plug paddles and joysticks as well as voltages which the computer can measure. Paddles usually consist of a box with a knob like a record player volume control. The computer can tell the position of the paddle and so it can be used in games and more serious programs to move things around the screen. Joysticks, on the other hand, can be moved left and right as well as up and down. As a result you can move an object anywhere on the screen not just up and down a particular line. Both paddles and joysticks can be fitted with push buttons and the computer can detect when these buttons are pressed. The BBC Microcomputer can be connected to four paddles or two joysticks. The BASIC function `ADVAL` can be used to detect the position of each control and of the fire buttons.

A second use for the analogue input is to measure voltages. Note that the analogue inputs have no built-in protection against excess voltages. You must therefore be careful not to apply a voltage greater than 1.8 volts or less than 0 volts to any of these inputs. In addition you should keep leads plugged into the analogue inputs away from devices which produce large static voltages such as televisions and some other mains equipment. Each of the four inputs can accept voltages in the range 0 to 1.8V and will produce a corresponding number in the range 0 to 65520. Since it is possible to use a transducer to produce a voltage proportional to temperature, light intensity, smoke density, water pressure, gas concentration etc, it is possible to use the computer to monitor all these things. If the unit is to be used to measure absolute voltages then it should be calibrated individually. In practice 1.0V input typically produces a reading of 35168. Although the unit is fitted with a 12 bit converter, the user should not rely on more than 10 bit accuracy unless great care is taken with screening and analogue ground connections.

Digital input/ouput using the eight bit user port

The BBC Microcomputer contains an eight bit user port which can be connected to a wide range of devices such as bit pads and general interfacing boxes. The user port can be read from or written to in BASIC and in assembly language, but in either case the user will need to know how to use the 6522 versatile interface adapter integrated circuit. A 6522 data sheet will be essential and the user will discover that this extremely versatile chip is also quite difficult to master. What follows is essential information that you will need to work the chip rather than a course in using it. Once you have learned to use it you will realise that at least 20 pages would be needed to give a decent introduction to it!

The 6522 lives in the memory map between locations &FE60 and &FE6F. The A side is used for the parallel printer port and the B side is used for the user port. The timers and shift register are also available for the user. When writing small programs the user can address the device directly either in BASIC or in Assembly Language. However programs that address the device directly will not work on the far side of the Tube. Machine code calls are provided to address the device whichever side of the Tube the program is on. Firstly, though, here are some programs in BASIC and assembly language to read and write to the port.

```
10 REM Read data in
20 REM Set Data Direction Register B
30 REM for all inputs
40 ?&FE62=0
50 REM read a value in and PRINT it
60 X=?&FE60
70 PRINT X
80 GOTO 60
```

The next program sets up the 6522 to output to the user port and then transfers the bottom eight bits of X to the user port. Again the initialisation need only take place once.

```
10 REM All outputs
20 ?&FE62=&FF
30 REM now put X out
40 ?&FE60=X
```

And here are those two programs in assembly language. First to read data into the accumulator:

```
100 LDA #0
110 STA &FE62
120 LDA &FE60
```

and secondly to write data out to the user port. This time the program is presented as two subroutines. The first, called **INIT**, sets up the 6522 and the second subroutine, **WRITE**, actually puts the data out from the accumulator onto the user port.

```
200 .INIT LDA #&FF
210 STA &FE62
220 RTS
230 .WRITE STA &FE60
240 RTS
```

As has been made clear above, these programs will not work from the second processor. The 6522 is one of the memory input/output devices in the area of memory referred to as SHEILA. SHEILA controls the section of memory map in the range &FE00 to &FEFF, and the VIA (versatile interface adaptor) uses addresses between &FE60 and &FE6F which are therefore SHEILA+&60 to SHEILA+&6F. Two OSBYTE calls (see chapter 43) are provided to read and write to SHEILA. Here are the same two routines shown above but written so that they *will* work over the Tube.

```
100 LDA #&97 \OSBYTE to write to SHEILA
110 LDX #&62 \Offset to Data direction reg.
120 LDY #0   \Value to be written
130 JSR &FFF4 \Call OSBYTE
140 LDA #&96  \OSBYTE to read from SHEILA
150 LDX #&60  \Offset to data register
160 JSR &FFF4 \Call OSBYTE to get value
```

And the next routine to **INIT** and **WRITE** to the user port:

```
200 .INIT LDA #&97 \OSBYTE to write to SHEILA
210 LDX #&62  \Offset to Data direction register
220 LDY #&FF  \All outputs
230 JSR &FFF4 \Call OSBYTE
240 RTS
250 .WRITE TAY  \Move value to Y
260 LDA #&97  \Write-to-SHEILA code
270 LDX #&60  \Offset to data register
280 JSR &FFF4 \OSBYTE call
290 RTS
```

In practice the user will often wish to use the handshake lines with data transfers. For information on this topic you are referred to other books. Space simply does not permit an adequate explanation here.

47 Error messages

If the computer is unable to proceed for some reason then it will report the fact to you by printing an error message on the screen. The printing of the message can be suppressed by an **ON ERROR statement** – for example

```
ON ERROR PROCerror
```
ON ERROR may be followed by any statement or multiple statement.

As well as the error message, the computer sets two variables each time an error occurs.

ERR gives the error number.

ERL gives the number of the line in the program where the error occurred.

REPORT is a command to print the last error message. For example

```
ON ERROR REPORT:PRINT" at line ";ERL:END
```

will give the same response that the computer gives without an **ON ERROR** statement.

The error messages are listed below in alphabetical order together with their error numbers.

Accuracy lost   23

If you try to calculate trigonometric functions with very large angles you are liable to lose a great deal of accuracy in reducing the angle to the range of plus or minus PI radians. In this case the computer will report Accuracy lost, eg

```
PRINT SIN(10000000)
```

Arguments   31

This error indicates that their are too many or too few arguments for a given function or procedure.

Array   14

This indicates that the computer thinks that an array is to be accessed but does not know the array in question.

Bad call   30

This indicates that the use of **PROC** or **FN** to call a defined procedure or function is incorrect.

Bad DIM 10

Arrays must be dimensioned with a positive number of elements. An error will be produced, for example, by:

```
DIM A(-3)
```

Bad HEX 28

Hex numbers can only include 0 to 9 and A to F. An attempt to form a hex number with other letters will result in this error, eg

```
PRINT &y
```

Bad key 251

An attempt has been made to provide a function key defenition with a key number greater than 15.

Bad MODE 25

This indicates an attempt to change mode inside a procedure or function, or to select a mode for which their is insufficient memory.

Bad program 0

There are a number of occasions on which the computer checks to see if the program that it contains starts and ends in memory. The untrappable and fatal error `Bad program` indicates that the computer could not follow a program through successfully to an end mark in memory. This is caused by a read error or by loading only part of a program or by overwriting part of a program in some way. Unless you are prepared to check the contents of memory a byte at a time there is little that can be done to recover a bad program.

Bad string 253

A string more than ten characters long has been passed with an operating system command (cassette filing system only).

Block? 218

This is an error generated by the cassette filing system. It indicates that an unexpected block number was encountered. Rewind the tape a short way and play it again Sam.

Byte 2

An attempt was made, during an assembly language section, to load a register with a number requiring more than one byte, eg

```
LDA #345
```

Can't match FOR 33

There is no **FOR** statement corresponding to the **NEXT** statement.

Channel 222

This error is generated by the cassette filing system if an attempt is made to use a channel that was not opened.

Data? 216

This an error generated by the cassette filing system and it means that the computer has found a cyclic redundancy check (CRC) error. The CRC is stored on tape along with other information. Rewind the tape a short way and play it again.

DIM space 11

An attempt was made to dimension an array for which there was insufficient room.

Division by zero 18

Division cannot be done, eg

```
PRINT 34/0
```

This error can also be caused by a division within a procedure or function using a **LOCAL** variable which has not been set to a new value. When a variable is declared as **LOCAL** it is set to zero.

$ range 8

The user may put strings into any place in memory except zero page – that is locations with addresses lass than &100. Thus this is illegal:

```
$40="hello"
```

Eof 223

This error is generated by the cassette filing system when the end of the file is reached.

Escape 17

The **ESCAPE** key has been pressed.

Exp range 24

The function **EXP** cannot deal with powers greater than 88. Thus the following is illegal:

```
X=EXP(90)
```

Failed at <line number>

When renumbering a program the computer attempts to look after all references made by **GOTO** and **GOSUB** statements. Thus the program

```
133 GOTO 170
170 END
```

would become

```
10 GOTO 20
20 END
```

when renumbered. However, the computer will not be able to deal with

```
133 GOTO 140
200 END
```

If the user attempts to renumber this program he or she will get the error message

```
Failed at line 10
```

and the renumbered program will be

```
10 GOTO 140
20 END
```

File? 219

This error indicates that an unexpected file name was encountered by the computer.

FOR variable 34

The variable in a **FOR...NEXT** loop must be a numeric variable. Thus the following is illegal:

```
FOR 5=3 TO 10
```

Header? 217

This an error generated by the cassette filing system and it indicates that a header cyclic redundancy check error has occurred. Rewind the tape a short way and play it again.

Index 3

This indicates an error in specifying an index mode when using the assembler, eg

```
LDA Z,Z
```

Key in use 250

An attempt has been made to define a function key while another function key is being expanded, eg

```
*KEY 0 *KEY 1 RUN |M |M
```

followed by pressing **f0** would produce this error message.

LINE space

The computer has no room left to insert the line in the program.

Log range 22

An attempt was made to calculate the **LOG** of a negative number or of zero, eg

```
PRINT LOG(-10)
```

Missing , 5

This error indicates that the computer expected to find a comma in the line, and didn't do so, eg

```
D$=MID$(A$)
```

Missing " 9

The computer expected to find a double quote, eg

```
LOAD "FRED
```

Missing) 27

The computer expected to find a closing parenthesis, eg

```
PRINT TAB(10,10
```

Missing #   45

The computer expected to find a #, eg

```
A=BGET
```

Mistake   4

This indicates that the computer could not make any sense of the input line.

-ve root   21

An attempt was made to calculate the square root of a negative number, eg

```
PRINT SQR(-10)
```

This may also occur with **ASN** and **ACS**.

No FN   7

This indicates that the computer detected the end of a function but had not called a function defenition, eg

```
=FNlinda
```

No FOR   32

A **NEXT** statement was found when no **GOSUB** statement had been encountered.

No GOSUB   38

A **RETURN** statement was found when no **GOSUB** statement had been encountered.

No PROC   13

This indicates that the word **ENDPROC** was found without there being a corresponding **DEF PROC** statement.

No REPEAT   43

The interpreter found an **UNTIL** statement when no **REPEAT** statement had been encountered.

No room 0

This untrappable and fatal error indicates that while the computer was running a program it used up all available memory.

No such FN/PROC 29

If the interpreter meets a name beginning with **FN** (eg **FNfred**) or **PROC** (eg **PROCrob**) it expects to find a corresponding function or procedure defenition somewhere. This error indicates that no matching defenition was found.

No such line 41

The computer was told to **GOTO** or **GOSUB** a line number which does not exist.

No such variable 26

All variables must be assigned to or made **LOCAL**, before they can be accessed in **PRINT** statements or before their values can be assigned to other variables. The initial assignment can simply be, for example, **X=0**.

No TO 36

A **FOR...NEXT** loop has been set up with the **TO** part missing. A correctly formed line is shown below.

```
FOR X= 10 TO 55
```

Not LOCAL 12

This indicates the appearance of **LOCAL** outside a procedure or function.

ON range 40

The control variable was either less than 1 or greater than the number of entries in the **ON** list. For example, if X=3 then the following will fail:

```
ON X GOTO 100,200
```

since there are only two destinations.

ON syntax   39

The `ON...GOTO` statement was incorrectly formed. For example, the following is illegal:

```
ON X PRINT
```

The word `ON` must be followed by a numeric which must in turn be followed by the word `GOTO` or `GOSUB`.

Out of DATA   42

An attempt was made to read more items of `DATA` than there were in the `DATA` list. The word `RESTORE` can be used to reset the data pointer to the start of the `DATA` if required.

Out of range   1

An attempt was made to branch out of range of the branch instruction in an assembly language program.

Silly

This message will be issued if you attempt to renumber a program or enter `AUTO` mode with a step size of 0 or more than 255, eg

```
AUTO 100,0
```

Syntax   220

This error is generated by the cassette filing system and indicates that a syntax error, such as an illegal `*OPT` statement has occurred.

String to long   19

The maximum length of a string is 255 characters.

Subscript   15

This implies that an attempt was made to access an element of an array less than zero or greater than the size of the array. For example, these two lines together will produce this error:

```
100 DIM A(10)
120 A(15)=3
```

Syntax error 16

A command was terminated wrongly, for example

```
LIST PRINT
```

Too big 20

A number was entered or calculated which was too large for the computer to handle.

Too many FORs 35

An attempt was made to nest too many `FOR...NEXT` loops. The maximum nesting allowed is ten. This can sometimes be caused by returning to a `FOR` statement without executing a `NEXT` statement, eg

```
10 FOR X=1 TO 6
20 GOTO 10
```

Too many GOSUBs 37

An attempt was made to nest too many `GOSUB...RETURN` loops. The maximum nesting allowed is 26. This can somtimes be caused by returning to a `GOSUB` statement without executing a `RETURN` statement, eg

```
10 PRINT "WRONG"
20 GOSUB 10
```

Too many REPEATS 44

An attempt was made to nest too many `REPEAT...UNTIL` loops. The maximum nesting allowed is 20. This can sometimes be caused by returning to a `REPEAT` statement without executing an `UNTIL` statement, eg

```
10 REPEAT
20 GOTO 10
```

Type mismatch 6

This error indicates that a number was expected and a string was offered or vice versa, eg

```
10 A$=X
```

Error number	Error message
1	Out of range
2	Byte
3	Index
4	Mistake
5	Missing ,
6	Type mismatch
7	No FN
8	$ range
9	Missing "
10	Bad DIM
11	Dim space
12	Not LOCAL
13	No PROC
14	Array
15	Subscript
16	Syntax error
17	Escape
18	Division by zero
19	String too long
20	Too big
21	-ve root
22	Log range
23	Accuracy lost
24	Exp range
25	Bad MODE
26	No such variable
27	Missing)
28	Bad HEX
29	No such FN/PROC
30	Bad call
31	Arguments
32	No FOR
33	Can't match FOR
34	FOR variable
35	Too many FORs
36	No TO
37	Too many GOSUBs
38	No GOSUB
39	ON syntax

40	**ON range**
41	**No such line**
42	**Out of DATA**
43	**No REPEAT**
44	**Too many REPEATs**
216	**Data?**
217	**Header?**
218	**Block?**
219	**File?**
220	**Syntax**
222	**Channel**
223	**Eof**
250	**Key in use**
251	**Bad key**
253	**Bad string**
254	**Bad command**

Note: disc filing system errors are described in the appropriate disc filing system user guide .

48 Minimum abbreviations

This chapter lists the minimum abbreviations that can be used for BASIC keywords. The third column lists the hexadecimal number that is used to store the keyword in memory. This is often referred to as the 'token'.

Notice that the abbreviation never needs an opening parenthesis because the token includes the parenthesis

ABS	ABS	94	ENDPROC	E.	E1
ACS	ACS	95	ENVELOPE	ENV.	E2
ADVAL	AD.	96	EOR	EOR	82
AND	A.	80	EOF	EOF	C5
ASC	ASC	97	ERL	ERL	9E
ASN	ASN	98	ERR	ERR	9F
ATN	ATN	99	ERROR	ERR.	85
AUTO	AU.	C6	EVAL	EV.	A0
BGET	B.	9A	EXP	EXP	A1
BPUT	BP.	D5	EXT	EXT	A2
CALL	CA.	D6	FALSE	FA.	A3
CHAIN	CH.	D7	FN	FN	A4
CHR$	CHR.	BD	FOR	F.	E3
CLEAR	CL.	D8	GCOL	GC.	E6
CLG	CLG	DA	GET	GET	A5
CLOSE	CLO.	D9	GET$	GE.	BE
CLS	CLS	DB	GOSUB	GOS.	E4
COLOUR	C.	FB	GOTO	G.	E5
COS	COS	9B	HIMEM	H.	93
COUNT	COU.	9C			(right)
DATA	D.	DC	HIMEM	H.	D3
DEF	DEF	DD			(left)
DEG	DEG	9D	IF	IF	E7
DELETE	DEL.	C7	INKEY	INKEY	A6
DIM	DIM	DE	INKEY$	INK.	BF
DIV	DIV	81	INPUT	I.	E8
DRAW	DR.	DF	INSTR(	INS.	A7
ELSE	EL.	8B	INT	INT	A8
END	END	E0	LEFT$(	LE.	C0
LEN	LEN	A9	PTR	PT.	CF
LET	LET	E9			(left)
LINE	LIN.	86	RAD	RAD	B2
LIST	L.	C9	READ	REA.	F3

LN	LN	AA	REM	REM	F4
LOAD	LO.	C8	RENUMBER	REN.	CC
LOCAL	LOC.	EA	REPEAT	REP.	F5
LOG	LOG	AB	REPORT	REPO.	F6
LOMEM	LOM.	92	RESTORE	RES.	F7
		(right)	RETURN	R.	F8
LOMEM	LOM.	D2	RIGHT$(	RI.	C2
		(left)	RND	RND	B3
MID$(	M.	C1	RUN	RUN	F9
MOD	MOD	83	SAVE	SA.	CD
MODE	MO.	EB	SGN	SGN	B4
MOVE	MOV.	EC	SIN	SIN	B5
NEW	NEW	CA	SOUND	SO.	D4
NEXT	N.	ED	SPC	SPC	89
NOT	NOT	AC	SQR	SQR	B6
OFF	OFF	87	STEP	S.	88
OLD	O.	CB	STOP	STO.	FA
ON	ON	EE	STR$	STR.	C3
OPENIN	OP.	8E	STRING$(	STRI.	C4
OPENOUT	OPENO.	AE	TAB(	TAB(	8A
OPENUP	OPENUP	AD	TAN	T.	B7
OPT	OPT		THEN	TH.	8C
OR	OR	84	TIME	TI.	91
OSCLI	OSC.	FF			(right)
PAGE	PA.	90	TIME	TI.	D1
		(right)			(left)
PAGE	PA.	D0	TO	TO	B8
		(left)	TRACE	TR.	FC
PI	PI	AF	TRUE	TRUE	B9
PLOT	PL.	F0	UNTIL	U.	FD
POINT(	PO.	B0	USR	USR	BA
POS	POS	B1	VAL	VAL	BB
PRINT	P.	F1	VDU	V.	EF
PROC	PRO.	F2	VPOS	VP.	BC
PTR	PT.	8F	WIDTH	W.	FE
		(right)			

49 BASIC II

This chapter details differences between BASIC II and the original version of BBC BASIC, and as such is mainly intended for those who are already familiar with the latter language.

Two new keywords are introduced by BASIC II: **OPENUP** and **OSCLI**. These are described in chapter 33.

BASIC II makes available the following alterations and extensions to existing BASIC keywords.

ABS
The unary minus operator may be used, for example

```
PRINT -ABS(-1)
```

will give the value -1. In BASIC this gave a Type mismatch error.

COUNT
This has been altered so that **COUNT** is reset to zero after a change of **MODE** , as shown by the following example program:

```
10 PRINT "Hello";
20 MODE 3
30 PRINT "Goodbye";
40 PRINT COUNT
```

In BASIC this would leave the screen showing:

```
Goodbye       12
```

With BASIC II the following is obtained:

```
Goodbye        7
```

ELSE
In BASIC, ON...GOTO...ELSE or **ON...GOSUB...ELSE** could not be used inside procedures or functions – only the ON...GOTO (or **ON...GOSUB**) part was available. This limitation is not present in BASIC II.

EVAL
In addition to its BASIC functions, in BASIC II **EVAL** can be used to evaluate the pseudo-variables **COUNT**, **ERL**, **ERR**, **HIMEM**, **LOMEM**, **PAGE**, **TIME** and **TOP**.

INPUT

If more than one string or value is to be input at a time then the variable identifiers have to be separated from each other. In BASIC this was done by using commas, eg

```
INPUT NAME$, AGE, HEIGHT
```

In BASIC II either commas or semicolons may be used, eg

```
INPUT NAME$, AGE; HEIGHT
```

When entering numerics however, these should be separated by commas as before, not semicolons.

INSTR

In BASIC II, `INSTR` has been extended such that an instruction such as

```
PRINT INSTR("l","Hello")
```

will return the value 0. This is useful in that a statement such as

```
X=INSTR(A$,B$)
```

can be included inside a procedure or function without having to check that `A$` is longer than `B$`.

ON ERROR

In BASIC II, the comand `ON ERROR GOTO...` can be used with any line number. (In BASIC, `ON ERROR GOTO 9999`, for example, could not be used).

OPENIN and OPENUP

In both BASIC and BASIC II an existing file can be opened to allow data to be read or altered, or to allow more data to be added to the end. In BASIC, this function was performed by the instruction `OPENIN`. In BASIC II it is done by `OPENUP`. Since these keywords give exactly the same result, the token for them both is &AD. Hence, if a program containing the instruction `OPENUP` is written on a BBC Microcomputer containing BASIC II then the instruction will be tokenised to &AD. If this program is then saved and loaded into a machine containing BASIC, the program will work in exactly the same way, but when listed it will display the instruction as `OPENIN`. This will apply the other way around as well so existing programs do not need to be altered to run in BASIC II.

The keyword **OPENIN** does exist in BASIC II, but has a different meaning. BASIC II uses the keyword **OPENIN** to open a file for read-only operations; this was not possible in BASIC. Since this is a new facility it has a new token, &8E. Note that programs written in BASIC II which contain the instruction **OPENIN** will not run in BASIC.

The following new features are available to assembly language programmers:

ASC
In BASIC II **ASC ":"** may be used in the assembler. In the original BASIC this lead to confusion.

EQUB, EQUD, EQUS, EQUW
These new features are detailed in chapter 44.

OPT
In BASIC II, bit 2 of the **OPT** statement's operand is used to determine whether assembled machine code is placed in memory at the address given by O% (the code origin) or P% (the program counter). See chapter 44 for full details.

478

Appendix A

Teletext (MODE 7) displayed alphanumeric characters

	0	10	20	30	40	50	60	70	80	90	100	110	120
0	Nothing	Down	Nothing	Move cursor to 00									
1	Next to printer	Up	Disable VDU	Move cursor									
2	Start printer	Clear screen	Select mode										
3	Stop printer	Start of line	Reprogram characters										
4	Nothing	Paged mode	Nothing										
5	Nothing	Scroll mode	Nothing										
6	Enable VDU	Nothing	Nothing										
7	Beep	Nothing	Nothing										Back space and delete
8	Back	Nothing	Nothing										Nothing
9	Forward	Nothing	Nothing										Alpha red

130	140	150	160	170	180	190	200	210	220	230	240	250
Alpha green	Normal * height	Graphic cyan										
Alpha yellow	Double height	Graphic white										
Alpha blue	Nothing	Conceal display										
Alpha magenta	Nothing	Contiguous graphics *										
Alpha cyan	Nothing	Separated graphics										
Alpha * white	Graphic red	Nothing										
Flash	Graphic green	Black * background										
Steady *	Graphic yellow	New background										
Nothing	Graphic blue	Hold graphics										
Nothing	Graphic magenta	Release * graphics										

* every line starts with these options

Appendix B

Teletext (MODE 7) displayed graphics characters

	0	10	20	30	40	50	60	70	80	90	100	110	120
0	Nothing	Down	Nothing	Move cursor to 00									
1	Next to printer	Up	Disable VDU	Move cursor									
2	Start printer	Clear screen	Select mode										
3	Stop printer	Start of line	Reprogram characters										
4	Nothing	Paged mode	Nothing										
5	Nothing	Scroll mode	Nothing										
6	Enable VDU	Nothing	Nothing										
7	Beep	Nothing	Nothing										Back space and delete
8	Back	Nothing	Nothing										Nothing
9	Forward	Nothing	Nothing										Alpha red

Each character has a code. Thus H is code 72 since it is in column 70 row 2.

130	140	150	160	170	180	190	200	210	220	230	240	250
Alpha green	Normal * height	Graphic cyan										
Alpha yellow	Double * height	Graphic white										
Alpha blue	Nothing	Conceal display										
Alpha magenta	Nothing	Contiguous graphics *										
Alpha cyan	Nothing	Separated graphics										
Alpha * white	Graphic red	Nothing										Back space and delete
Flash	Graphic green	Black * background										
Steady *	Graphic yellow	New background										
Nothing	Graphic blue	Hold graphics										
Nothing	Graphic magenta	Release graphics *										

* every line starts with these options

Appendix C

ASCII (MODES 0 to 6) displayed character set and control codes

	0	10	20	30	40	50	60	70	80	90	100
0	Nothing	Down	Default logical colour	Move text cursor to 00	(	2	<	F	P	Z	d
1	Next to printer	Up	Disable VDU	Move text cursor	)	3	=	G	Q	[	e
2	Start printer	Clear text	Select mode		*	4	>	H	R	\	f
3	Stop printer	Start of line	Reprogram characters	!	+	5	?	I	S	]	g
4	Separate cursors	Paged mode	Define graphics area	"	,	6	@	J	T	^	h
5	Join cursors	Scroll mode	Plot	#	-	7	A	K	U	_	i
6	Enable VDU	Clear graphics	Default text/ graphics areas	$	.	8	B	L	V	£	j
7	Beep	Define text colour	Nothing	%	/	9	C	M	W	a	k
8	Back	Define graphics colour	Define text area	&	0	:	D	N	X	b	l
9	Forward	Define logical colour	Define graphics origin	'	1	;	E	O	Y	c	m

110 120 130 140 150 160 170 180 190 200 210 220 230 240 250

n x

o y

p z

q {

r H

s }

t

All characters
undefined initially

u

Back
space
and
delete

v

w

Appendix D

Hexadecimal codes

MSB LSB	8 / 0	9 / 1	A / 2	B / 3	C / 4	D / 5	E / 6	F / 7
0000 0	Nothing	Clear graphics area		∅	@	P	£	p
0001 1	Next to printer	Define text colour	!	1	A	Q	a	q
0010 2	Start printer	Define graphic colour	"	2	B	R	b	r
0011 3	Stop printer	Define logical colour	#	3	C	S	c	s
0100 4	Separate cursors	Default logical colour	$	4	D	T	d	t
0101 5	Join cursors	Erase line or disable VDU	%	5	E	U	e	u
0110 6	Enable VDU	Select mode	&	6	F	V	f	v
0111 7	Beep	Reprogram characters	'	7	G	W	g	w
1000 8	Back	Define graphics area	(	8	H	X	h	x
1001 9	Forward	Plot	)	9	I	Y	i	y
1010 A	Down	Default screen areas	*	:	J	Z	j	z
1011 B	Up	Nothing	+	;	K	[	k	{
1100 C	Clear text area	Define text area	,	<	L	\	l	|
1101 D	Carriage return	Define graphic origin	-	=	M	]	m	→
1110 E	Paged mode on	Move text cursor to ∅,∅	.	>	N	^	n	~
1111 F	Paged mode off	Move text cursor to X, Y	/	?	O	—	o	Back space and delete

Appendix E

Text and graphics planning sheets

Graphics planning sheet 1 (grid related to character positions)

Graphics planning sheet 2 (decimal)

128 64 32 16 8 4 2 1

User defined character planning sheet

MODES 0 to 6

Appendix F

Keyboard codes

Most keys can produce three codes and these are shown for each key. The top number is the code produced if the **CTRL** key is simultaneously depressed. The middle number is the upper case code and the lower number the lower case code. Thus

CTRL Z is &1A

Z is &5A

z is &7A

All numbers are in hexadecimal.

The editing keys only produce codes if enabled with * F X 4.

Appendix G

Printed circuit board layout for the BBC Microcomputer

Appendix H

External connections at the rear of the BBC Microcomputer

Appendix I

External connections underneath the BBC Microcomputer.

Tube

1MHz bus

User input/output

Printer

Disc

Appendix J

Memory map and memory map assignments

Memory map

	Hex	Decimal
Operating system ROM	&FFFF	65535
Memory mapped input/output	&FF00	65280
	&FC00	64512
Operating system ROM		
	&C000	49152
- - - - - - - - - -	&AFFF	45055 - - -
Four paged ROMs eg BASIC		Paged RAM (12K)
	&8000	32768 - -
RAM used for high resolution graphics		
HIMEM →	Movable boundary	
BASIC stack		
		Shadow screen (20K)
32K RAM to &8000	&4000	16384
- - - - - -	&3000	12288 - - -
LOMEM → Dynamic variable storage	Movable boundary	
TOP →	&2000	8192
User's BASIC program area		
PAGE →	&0E00	3584
Reserved for operating system use		
	&0000	0

Reserved	&EØØ	3584
User defined character definitions	&DØØ	3328
User defined function key definitions	&CØØ	3Ø72
Various buffers	&BØØ	2816
Various buffers	&AØØ	256Ø
Miscellaneous workspace	&9ØØ	23Ø4
Language ROM workspace	&8ØØ	2Ø48
Miscellaneous workspace	&4ØØ	1Ø24
Operating system workspace	&3ØØ	768
6512 stack	&2ØØ	512
Zero page	&1ØØ	256
	&ØØØØ	Ø

Memory map (detail)

Memory map assignments

FF00 to FFFF	Operating System ROM.
FE00 to FEFF	Internal memory mapped input/output (SHEILA).
FD00 to FDFF	External memory mapped input/output (JIM).
FC00 to FCFF	External memory mapped input/output (FRED).
C000 to FBFF	Operating System ROM.
8000 to BFFF	One or more sideways ROMs (eg BASIC, VIEW, BCPL, PASCAL).
0000 to 7FFF	Read/write RAM.
1900 to 1AFF	Econet filing system workspace (if fitted).
E00	Default setting of **PAGE**.
E00 to 18FF	Disc filing system workspace (if fitted).
E00 to 1CFF	Advanced disc filing system workspace (if fitted).
D9F to DFF	ROM workspace.
D00 to D9E	Used by NMI routine (eg by disc or Econet filing system).
C00 to CFF	User defined character definitions.
B00 to BFF	User defined function key (soft key) definitions.
A00 to AFF	RS423 receive.
900 to 9FF	RS423 transmit, sound and speech workspace.
800 to 8FF	Sound and buffer workspace.
400 to 7FF	Sideways ROM workspace.
300 to 3FF	VDU, cassette and keyboard workspace.
200 to 2FF	Operating system workspace and indirection vectors.
100 to 1FF	6512 stack.
000 to 0FF	Zero page.

Zero page

FF	The top bit is set during an ESCAPE condition.
FD to FE	Address following detected **BRK** instruction.
FC	User IRQ routine save slot for A register.
D0 to FB	Used by machine operating system.
B0 to CF	Allocated to current filing system.
A8 to AF	Used by machine operating system.
A0 to A7	Allocated to disc or Econet filing system.
90 to 9F	Allocated to Econet filing system.
70 to 8F	Free for user routines (in BASIC only).
0 to 6F	BASIC language (or currently selected ROM).

Sideways (shadow) RAM

8000 to AFFF	Paged RAM
3000 to 7FFF	Shadow screen RAM

Appendix K

Circuit layouts

Printer, User I/O, 1 MHz Bus and Tube circuits

Video outputs

RS423 interface

Analogue inputs

CASSETTE
CONNECTOR
SK5

R151
10K

+5V

D17
1N 4148

D18
1N 4148

0V

C58
820pF

C59
820pF

R145
150K

0V

LM 324
IC 89

R135
820K

R136
220K

R137
150K

R127
8K2

R123
8K2

0V

0V

RL 1

0V

R17
39K

LM 324
IC 89

C43
4n7F

0V

C42
4n7F

+5V

D16
1N 4148

C41
220nF

R111
10K

R103
5K6

−5V

Q7
BC 239

7

LM 324
IC 89

5

6

C38
220nF

R100
4K7

CASS
LED

0V

Cassette interface

Appendix L

VDU code summary

Decimal	Hex	CTRL	ASCII abbreviation	Bytes extra	Meaning
0	0	@	NUL	0	Does nothing
1	1	A	SOH	1	Send next character to printer only
2	2	B	STX	0	Enable printer
3	3	C	ETX	0	Disable printer
4	4	D	EOT	0	Write text at text cursor
5	5	E	ENQ	0	Write text at graphics cursor
6	6	F	ACK	0	Enable VDU drivers
7	7	G	BEL	0	Make a short bleep
8	8	H	BS	0	Backspace cursor one character
9	9	I	HT	0	Forwardspace cursor one character
10	A	J	LF	0	Move cursor down one line
11	B	K	VT	0	Move cursor up one line
12	C	L	FF	0	Clear text area
13	D	M	CR	0	Move cursor to start of current line
14	E	N	SO	0	Page mode on
15	F	O	SI	0	Page mode off
16	10	P	DLE	0	Clear graphics area
17	11	Q	DC1	1	Define text colour
18	12	R	DC2	2	Define graphics colour
19	13	S	DC3	3	Define logical colour
20	14	T	DC4	4	Restore default logical colours
21	15	U	NAK	0	Disable VDU drivers or delete current line
22	16	V	SYN	1	Select screen mode
23	17	W	ETB	9	Reprogram display character
24	18	X	CAN	8	Define graphics character
25	19	Y	EM	5	`PLOT K,x,y`
26	1A	Z	SUB	0	Restore default windows
27	1B	[	ESC	0	Does nothing
28	1C	\	FS	4	Define text window
29	1D	]	GS	4	Define graphics origin
30	1E	^	RS	0	Home text cursor to top left
31	1F	_	US	2	Move text cursor to x, y
127	7F		DEL	0	Backspace and delete

Appendix M

6502 instruction set

MNEMONIC	OPERATION	IMMEDIATE	ABSOLUTE	ZERO PAGE	ACCUM	IMPLIED	(IND X)	(IND Y)	Z PAGE X	ABS X	ABS Y	RELATIVE	INDIRECT	Z PAGE Y	PROCESSOR STATUS CODES	MNEMONIC
ADC	A + M + C → A															ADC
AND	A ∧ M → A															AND
ASL	C ← 7 ... 0															ASL
BCC	BRANCH ON C=0															BCC
BCS	BRANCH ON C=1															BCS
BEQ	BRANCH ON Z=1															BEQ
BIT	A ∧ M															BIT
BMI	BRANCH ON N=1															BMI
BNE	BRANCH ON Z=0															BNE
BPL	BRANCH ON N=0															BPL
BRK	BREAK															BRK
BVC	BRANCH ON V=0															BVC
BVS	BRANCH ON V=1															BVS
CLC	0 → C															CLC
CLD	0 → D															CLD
CLI	0 → I															CLI
CLV	0 → V															CLV
CMP	A − M															CMP
CPX	X − M															CPX
CPY	Y − M															CPY
DEC	M − 1 → M															DEC
DEX	X − 1 → X															DEX
DEY	Y − 1 → Y															DEY
EOR	A ⊻ M → A															EOR
INC	M + 1 → M															INC
INX	X + 1 → X															INX
INY	Y + 1 → Y															INY
JMP	JUMP TO NEW LOC															JMP
JSR	JUMP SUB															JSR
LDA	M → A															LDA

		(1)																			M7		
LDX	M→X																						
LDY	M→Y																						
LSR	0→[ ]																						
NOP	NO OPERATION																						
ORA	A V M→A																						
PHA	A→Ms																						
PHP	P→Ms																						
PLA	Ms→A																						
PLP	Ms→P																						
ROL																							
ROR	C→	=																					
RTI	RTRN INT																						
RTS	RTRN SUB																						
SBC	A−M−C̄→A	(1)																					
SEC	1→C																						
SED	1→D																						
SEI	1→I																						
STA	A→M																						
STX	X→M																						
STY	Y→M																						
TAX	A→X																						
TAY	A→Y																						
TSX	S→X																						
TXA	X→A																						
TXS	X→S																						
TYA	Y→A																						

(1) ADD 1 to N IF PAGE BOUNDARY IS CROSSED
(2) ADD 1 TO N IF BRANCH OCCURS TO SAME PAGE
    ADD 2 TO N IF BRANCH OCCURS TO DIFFERENT PAGE
(3) CARRY NOT = BORROW
(4) IF IN DECIMAL MODE Z FLAG IS INVALID
    ACCUMULATOR MUST BE CHECKED FOR ZERO RESULT

X	INDEX X	+ ADD
Y	INDEX Y	− SUBTRACT
A	ACCUMULATOR	∧ AND
M	MEMORY PER EFFECTIVE ADDRESS	∨ OR
Ms	MEMORY PER STACK POINTER	⊻ EXCLUSIVE OR

M7 MEMORY BIT 7
M6 MEMORY BIT 6
n NO CYCLES
NO BYTES

Appendix N

*FX and OSBYTE call summary

Decimal	Hex	Function
0	0	Prints operating system version number.
1	1	Reserved for application programs.
2	2	Selects input device.
3	3	Selects output devices.
4	4	Enable/disable cursor edit keys.
5	5	Select printer type.
6	6	Set printer ignore character.
7	7	Set RS423 receive baud rate.
8	8	Set RS423 transmit baud rate.
9	9	Set flash period of first colour.
10	A	Set flash period of second colour.
11	B	Set auto-repeat delay.
12	C	Set auto-repeat period.
13	D	Disable various events.
14	E	Enable various events.
15	F	Clear all or just input buffer.
16	10	Select number of ADC channels.
17	11	Force start of conversion on ADC channel.
18	12	Reset user defined function keys.
19	13	Wait for field synchronisation.
20	14	Explode soft character RAM allocation.
21	15	Clear selected buffer.
114	72	Control shadow/main memory selection
117	75	Read VDU status byte.
118	76	Read **CTRL/SHIFT** key status.
119	77	Close ***SPOOL** and ***EXEC** files.
123	7B	End of user print routine.
124	7C	Reset ESCAPE flag.
125	7D	Set ESCAPE flag.
126	7E	Acknowledge detection of escape condition.
127	7F	Check end of file status.
128	80	Read ADC channel/fire buttons/last conversion.
129	81	Read key within time limit.
130	82	Read machine high order address.
131	83	Read top of operating system RAM address.
132	84	Read bottom of display RAM address.

133	85	Read lowest address for particular mode.
134	86	Read text cursor position.
135	87	Read character at text cursor position.
137	89	Turn cassette motor on/off.
138	8A	Insert character into specified buffer.
139	8B	Set file options.
140	8C	Select cassette file system and set speed.
142	8E	Select sideways ROM.
144	90	Alter TV display position/interlace.
145	91	Remove character from buffer.
146	92	Read from I/O area FRED.
147	93	Write to I/O area FRED.
148	94	Read from I/O area JIM.
149	95	Write to I/O area JIM.
150	96	Read from I/O area SHEILA.
151	97	Write to I/O area SHEILA.
152	98	Examine specified buffer.
158	9E	Read from speech processor.
159	9F	Write to speech processor.
209	D1	Speech on/off.
210	D2	Sound on/off.
218	DA	Read/write size of VDU queue
239	EF	Read/write shadow display mode state
224	E0	Cancel VDU queue.
225	E1	Set base number for function key codes.
226	E2	Set base number for **SHIFT** function key codes.
227	E3	Set base number for **CTRL** function key codes.
228	E4	Set base number for **SHIFT CTRL** function key codes.
229	E5	**ESCAPE**=&1B.
230	E6	Enable/disable normal **ESCAPE** key action.
231	E7	Enable/disable user 6522 IRQ.
232	E8	Enable/disable 6850 ACIA IRQ.
233	E9	Enable/disable system 6522 IRQ.
235	EB	Return presence of speech processor.
253	FD	Last reset type.
255	FF	Write start-up option byte.

Appendix O

Operating system calls

Routine		Vector		Summary of function
Name	Address	Name	Address	
		UPTV	222	User print routine
		EVNTV	220	Event interrupt
		FSCV	21E	File system control entry
OSWRSC	FFB3	–	–	Write byte to screen
OSRDSC	FFB9	–	–	Read byte from screen
OSFIND	FFCE	FINDV	21C	Open or close a file
OSGBPB	FFD1	GBPBV	21A	Load or save a block of memory to file
OSBPUT	FFD4	BPUTV	2	Save a single byte to file from A
OSBGET	FFD7	BGETV	216	Load a single byte to A from file
OSARGS	FFDA	ARGSV	214	Load or save data about a file
OSFILE	FFDD	FILEV	212	Load or save a complete file
OSRDCH	FFE0	RDCHV	210	Read character (from keyboard) to A
OSASCI	FFE3	–	–	Write a character (to screen) from A plus LF if (A)=&0D
OSNEWL	FFE7	–	–	Write LF,CR (&0A,&0D) to screen
OSWRCH	FFEE	WRCHV	20E	Write character (to screen) from A
OSWORD	FFF1	WORDV	20C	Perform miscellaneous OS operation control block to pass parameters
OSBYTE	FFF4	BYTEV	20A	Perform miscellaneous OS operation using registers to pass parameters
OSCLI	FFF7	CLIV	208	Interpret the command line given
		IRQ2V	206	Unrecognised IRQ vector
		IRQ1V	204	All IRQ vector
		BRKV	202	Break vector
		USERV	200	Reserved

Index