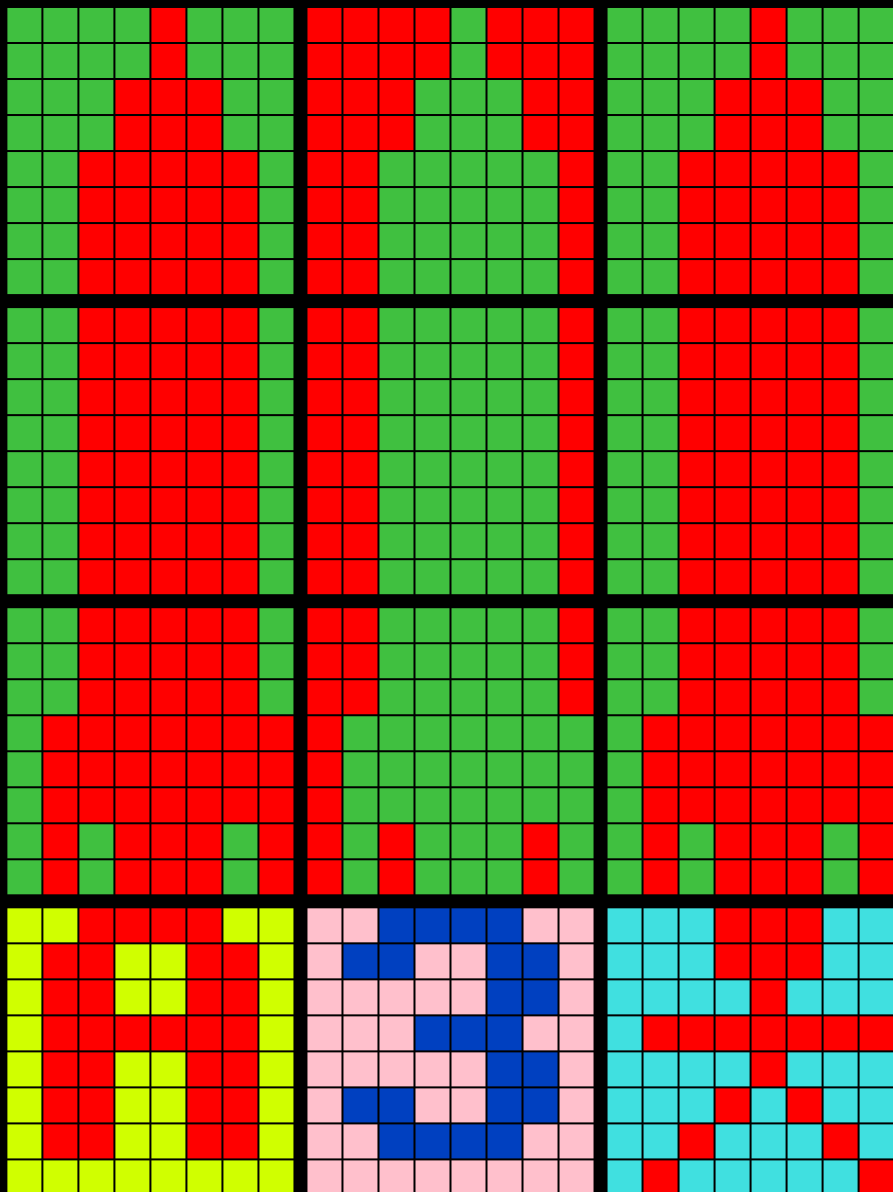




BRITISH BROADCASTING CORPORATION

MICROCOMPUTER SYSTEM

# USER GUIDE



# BBC Microcomputer User Guide

## Addendum Slip

### PLEASE READ THIS BEFORE USING THE WELCOME CASSETTE

If your machine includes Disc, Econet or Teletext Interface, your attention is drawn to page 400, describing how to change between filing systems.

It will be necessary, before loading the Welcome cassette programs (page 12) to change to the Tape Filing System (contained within the Machine Operating System Rom.). This is done by typing:

**\*TAPE RETURN**

before the

**CHAIN "WELCOME" RETURN**

command, and 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** [see entry under keywords  
for explanation of **PAGE**]

If at any time you wish to return to the Disc or Net filing systems, press **BREAK** or **CTRL BREAK**, then type:

**\*DISC RETURN**

or

**\*NET RETURN**

The BBC Microcomputer

# USER GUIDE

Written by John Coll

Edited by David Allen

British Broadcasting Corporation

## **WARNING: THE COMPUTER MUST BE EARTHED**

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

<b>Green and yellow</b>	<b>Earth</b>
<b>Blue</b>	<b>Neutral</b>
<b>Brown</b>	<b>Live</b>

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 would 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.

The author and editor would like to thank Paul Bond, Chris Charlesworth, Computer Concepts, Fiona Hibberd, Charles Moir, Jim Murray, Richard Russell, Fenella Sturt, Robin Viet and Roger Wilson for their invaluable help in the preparation and checking of the manuscript and Norman Brownsword the designer.

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.

© The Author and The British Broadcasting Corporation 1982

First published 1982

Remastered by dv8 in 2017

Fifth revision April 2023

The latest revision of this book is available at:

<https://stardot.org.uk/forums/viewtopic.php?t=14024>

Published by the British Broadcasting Corporation

35 Marylebone High Street, London W1M 4AA

ISBN 0 563 16558 8

# Contents

Introduction	5
1 Getting going	7
<b>Giving the computer instructions – Part I</b>	
2 Commands	21
3 An introduction to variables	24
4 Writing a simple program	27
5 Recording programs on cassette	34
6 Some sample programs	38
<b>Giving the computer instructions – Part II</b>	
7 AUTO, DELETE, REM and RENUMBER	53
8 Introducing Graphics	55
9 More on variables – string, real & integer ASCII codes, CHR\$ and ASC	62
10 PRINT and formatting. Cursor control	67
11 INPUT	78
12 GET and INKEY	81
13 TIME and Random numbers	84
<b>Structure in BASIC</b>	
14 REPEAT...UNTIL, TRUE and FALSE	87
15 FOR...NEXT	91
16 IF...THEN...ELSE	98
17 PROCEDURES	102
18 FUNCTIONS	110
19 GOSUB	113
20 GOTO, ON GOTO and ON GOSUB	118
<b>Giving the computer instructions – Part III</b>	
21 Yet more on variables – Arrays	120
22 READ, DATA and RESTORE	126
23 Integer handling	130
24 String handling	135
25 Programming the User Defined function keys	141
26 Operator precedence	144
27 Error handling	147
28 Use of teletext mode (MODE 7)	150

29	Advanced Graphics: (logical, inverse, actual colours), PLOT, GCOL, animated graphics, user definable characters	160
30	Sound	180
31	File handling	188
32	Speeding up programs and saving memory space	194

## Reference section

33	BASIC keywords alphabetical summary	197
34	VDU drivers	377
35	Cassette files	390
36	Changing filing systems	400
37	Merging BASIC Programs	402
38	Using printers	404
39	Indirection operators	409
40	HIMEM, LOMEM, TOP and PAGE	414
41	Operating system, statements (*commands)	416
42	*FX calls and OSBYTE calls	418
43	Assembly Language	442
44	Analogue input and user input-output ports	467
45	Expanding the system	471
46	Error messages	474
47	Minimum abbreviations	483
48	Appendix	485
	ASCII Teletext codes	486
	ASCII Teletext codes and shapes	488
	ASCII (MODES 0 to 6) displayed character set	490
	ASCII hexadecimal codes	492
	Text planning sheet	493
	Graphics planning sheets	494
	Keyboard codes	497
	Board layout	498
	External connections	499
	Memory maps	500
	Memory map assignments	502
	Circuit layouts	503
	VDU code summary	507
	6502 instruction set	508
	FX call summary	510
	Operating System call summary	512
	BASIC II keywords OPENUP and OSCLI	513
	<b>INDEX</b>	<b>517</b>

# Introduction

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

**BBC Microcomputer**

**Guarantee registration card**

**An aerial lead about 2 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 despatch 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 page 13). 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.

## **What this book can and can't do**

The **BBC Microcomputer** is a very versatile machine. On its own, connected to your television set, it can respond to programs which you yourself type in, to produce numbers, words, lines, movement and sound on the screen. 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 sixteen programs specially written

for the machine. Others will be made available in increasing numbers as time goes by. These will include programs linked to hobbies, games and programs for the home and for business and educational use.

The early sections of this book will show you how to load and save programs from tape, 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 (for details of courses write to the address given below).

Most of what follows in the later sections 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. To help matters, throughout the book we have used a second typeface to show what appears on your screen or what is typed in at the keyboard.

This book is not the last word on the BBC Microcomputer. Other more specialist books will appear describing the use of its more sophisticated features and how the machine can be expanded in various ways to make it increasingly useful. You will also find other 'fun' books appearing providing programs for you to type in yourself (in the same way that you can with the programs we include in this Guide). Pages on the BBC's Ceefax service will also be broadcast with programs for copying by hand into the machine. (It will also be possible to load these directly into the computer via the special Teletext decoder which will be made available as an extra in 1982. This facility is known as **Telesoftware** and is one of the most exciting possibilities opened up by the BBC Microcomputer.)

For details of courses in programming, books on the BBC system, software programs in the BBC Software List, including Telesoftware, write to:

**BBC Computer Literacy Project**  
**PO Box 7**  
**London W3 6XJ**

David Allen (Project Editor)



# 1 Getting going

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.

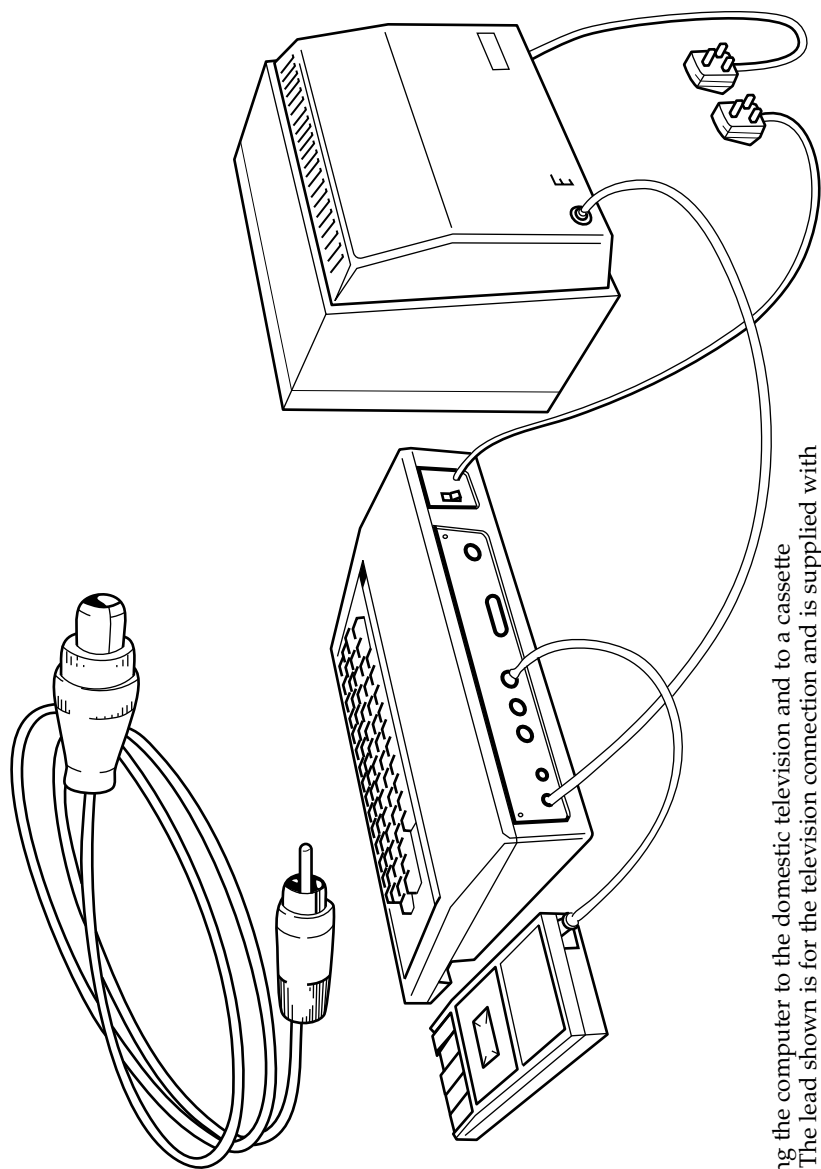
*If you have a high quality monitor (for example in a school) then it can be connected directly to one of the sockets on the back of the computer. To connect the monitor to the computer you will need a special monitor lead.*

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 (*figure 1*). Don't worry about the cassette recorder for the moment.

Next, plug your computer into the mains and switch it on. It should make a short 'beep' and the red light marked CAPS LOCK should come on. The switch on the back of the computer is marked ON/OFF. 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". Sometimes the screen will appear to be just blank. 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 BBC1, 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.



**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.

Different makes of television set 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

**BBC Computer 32K**

**BASIC**

**>**

should be clear and sharp. Many types of tuning control indicate, approximately, 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.

After that, 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

**M i s t a k e**

whenever you press the large key marked **RETURN**. That 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 aim to 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 **M i s t a k e** 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 0,1000**

**GCOL 0,1**

**PLOT 85,0,0**

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

**PLOT 86,1000,1000**

**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 will have gathered 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 rather 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 is 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 sounds as well. Try typing this in:

SOUND 1,-15,100,200

and then press **RETURN**

That gives a rather 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 effect.

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 section 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

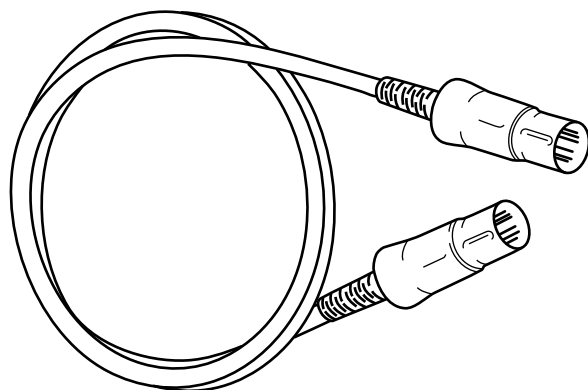
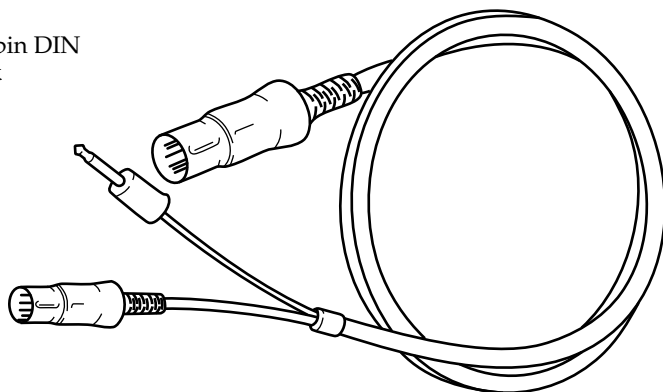
Press the **BREAK** key and type in the following, exactly as shown

```
CHAIN "WELCOME"
```

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:

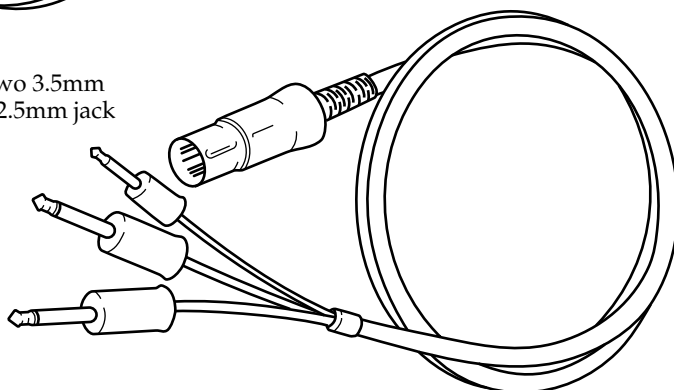
```
Your volume control is now properly  
set. Please wait while the first  
program is loaded
```

ANG0  
7 pin DIN to 5 pin DIN  
and 2.5mm jack



ANG0  
7 pin DIN to  
7 pin DIN

ANG0  
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 DIN 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.

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 internal 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 program 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 2 minutes (once the volume control is set) and then begin playing the tape from this point, having first entered the command **CHAIN "WELCOME"**.

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 sixteen programs does.



As time goes by, a whole range of 'application programs' will be available on cassette and it will be possible to run these in exactly the same way as the WELCOME package.

### The keyboard

Anyone who has used a standard typewriter will be reasonably 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 daunting. 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.

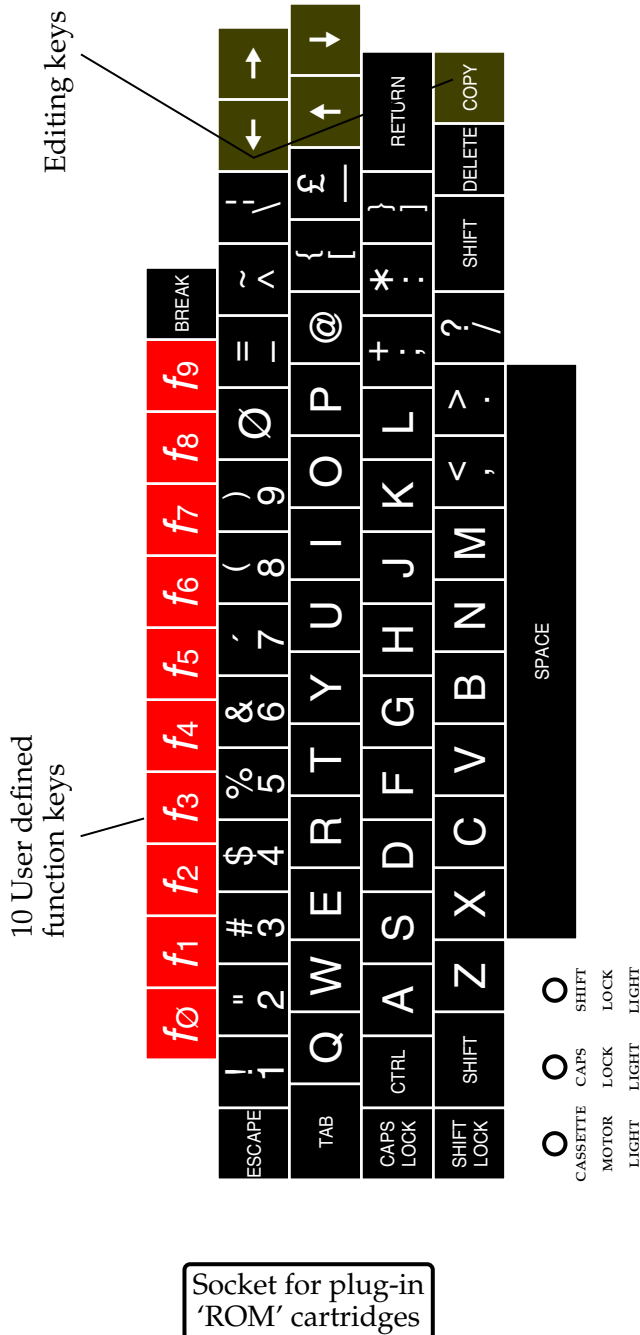
#### **CAPS LOCK**

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.



**Figure 3**  
The keyboard of the BBC Microcomputer.  
For details, see text

### **SHIFT LOCK**

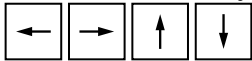
Pressing this key once gives capital letters and upper case symbols until it is pressed again. It has its own on/off light.

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

### **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 page 29).

### **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 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 to move 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.

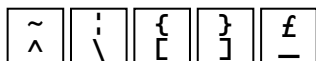
Do not get into the habit of using **BREAK**. The **ESCAPE** key provides a much less violent way of escaping, from a program! (See page 142 for more details on **BREAK**).

## **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 on page 378.

## **TAB**

Another key 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 Computer can display both. **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 it has to display on the screen. For completeness, here is a table showing all these characters:

On the key	Displayed on the screen in <b>MODE 7</b>	in <b>MODES</b> 0 to 6
~	÷	~
^	↑	^
:		:
\	<sup>1</sup> / <sub>2</sub>	\
{	<sup>1</sup> / <sub>4</sub>	{
[	←	[
}	<sup>3</sup> / <sub>4</sub>	}
]	→	]

Note that in **MODE 7** a zero is shown as a rather pointed 0 whereas in all other modes, zeros have a slash thus – 0 – to help

to differentiate them from the letter O. The keyboard is also marked in this way.

**Plug in cartridge socket**

There may be a slot to the left of the main keyboard where it will be possible to insert cartridge packs containing games and other programs. This inexpensive option will enable users to play games and use other programs with the minimum of effort.

# **Giving the computer instructions – Part I**

# 2 Commands

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

- 1 Give it **commands** which it can act on straight away. This is what happened when you typed in the lines in section 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 section 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 are all 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 things. They are:

- + addition
- subtraction
- \* multiplication
- / division
- ^ raise to the power
- . 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 quickly 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 in what you type in.

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 affect next time you press **BREAK** or change **MODE**.

# 3 An introduction to variables

In the last section 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 it to all the other wages that you had worked out so far, 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 any time you want to.

The computer is able to store hundreds of different variables and the variables don't just 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 write

**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, though you can stick as many numbers in as you want to later on.
- 3 You must not use punctuation marks (like 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 quite permissible to start a variable name with a lower case "to" because upper and lower case names are quite different. There is a full list of keywords starting on page 483.

To get lower case characters on the screen, make sure that the **CAPSLock** is off by depressing it so that its light goes out. In this condition you will get small letters and numbers. Hold the **SHIFT** key down if you want to get just a few capital letters.

Any of the following variable names would be acceptable to the computer

```
LET AGE=38
LET this_year=1982
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]
```

<code>LET TODAY=23</code>	[starts with <code>T0</code> ]
<code>LET PRINT=1234.56</code>	[ <code>PRINT</code> 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 “1982”. The word `LET` is optional and the computer will understand perfectly well if we say

`this_year=1982`

This shortened version is much more common.

# 4 A simple program

In the previous sections 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 5p, 10p and 50p coins 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 coins required.

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

Inevitably you would spend a great deal of time waiting for the mechanic to complete the job that you had set him before you could give him the next job. 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 headlamp bulb,  
thirdly stop the exhaust rattling.

Once you have given your set of instructions and checked that the garage understands what they have to do, you can then walk off and have a cup of coffee and then go back expecting the job to be done. Now the same thing applies with a computer. It is far better to give it a whole set of instructions

and set it going on something 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 straight away, but expect other instructions to follow. Instruction number 10 is usually referred to as Line 10 etc. 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** on the keyboard. It is 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** button.

When you do that it will stop running your program and print a > prompt to show that it has stopped the program and that you have command again.

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 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**.

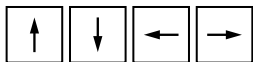
The computer will then give a list of the program on the screen for you to check. If you discover that you have made an error, for example that you have got something wrong in line 20, then it is easy to correct the error. There are two ways of correcting major errors:

- 1 Retype the whole line
- 2 Use the screen editor

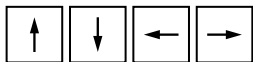
### Using the screen editor

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 than 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. Half way 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 been 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** becoming the space after **BIG**. At the end press **RETURN**.

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

- a List the program by using the **LIST** command.



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

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

c 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 in 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 go 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.

d 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!

### Removing 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 just want to delete a line completely then type in just the line number and press **RETURN** thus:

```
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 tidy sequence. Here is an awful example of 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 all the line numbers but it has accurately 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 a 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 the cassette. Also you must **leave gaps between programs**. It is very 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 10 characters. For example you could have typed*

**SAVE "FRED" or**

**SAVE "GAME3" or**

**SAVE "picture"**

While the program is being saved on the 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 page 36). If your recording failed for any reason you can always re-record it. See page 390 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"**

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 page 12 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** to make it work, as usual.

There is one more useful feature to do with loading and saving programs. Instead of typing **LOAD "MYPROG"** you can type **CHAIN "MYPROG"**. This not only loads in the program **MYPROG** but also starts it working as soon as it has loaded. It saves you having to type **RUN** after the program 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 page 71 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 on page 390.

# 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 as accurately as possible and can then be *run*. If a program fails to run properly, then almost certainly you have typed a line in incorrectly – for instance, you may have typed `;` when you should have typed `:` or typed `0` instead of `Ø`.

Most of the sample programs are too big to fit on the screen in one go. 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 Ø** 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 have satisfied yourself that they do run properly, will enable you to start to build 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 at once. 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 jig-saw 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 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 co-ordinates 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) thus making it appear that the new polygon is in front of older ones. Lines 310 to 370 draw all the lines that make up the polygon.

Lines 50 to 70 set the actual colours 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 MODE 5
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(1000)
```

```

140  VDU 29,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    MOVE 0,0
260    PLOT 85,X(SIDE),Y(SIDE)
270  NEXT SIDE
280  MOVE 0,0
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 twelve month period. In its present form it will only work on a Model B Computer. With minor modifications (eg changing line 100 to **MODE 5**) it can be made to run on Model A. 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 0 resets the computer to **MODE 7** and then *lists* the program. Key 9 can be used to *run* the program.

```

10 REM MONTHLY
20 REM JOHN A COLL
30 REM VERSION 1 / 16 NOV 81
40 REM MODEL B
50
60 *KEY 0 "MODE7 |M LIST|M"
70 *KEY 9 "RUN |M"
80 M$="JFMAMJJASOND"
90 C=0
100 MODE 2
110 VDU 5
120 VDU 29,0;100;
130
140 FOR X=0 TO 1100 STEP 100
150   GCOL 0,C MOD 7+1
160   C=C+1
170   H=RND(400)+300
180   MOVE X,0
190   MOVE X,H
200   PLOT 85,X+100,0
210   PLOT 85,X+100,H
220   MOVE X+70,H+50
230   MOVE X,H
240   PLOT 85,X+170,H+50
250   PLOT 85,X+100,H
260   PLOT 85,X+170,50
270   PLOT 85,X+100,0
280   GCOL 0,0
290   MOVE X,H
300   DRAW X+100,H
310   DRAW X+170,H+50
320   MOVE X+100,H
330   DRAW X+100,0
340   MOVE X+10,50
350   PRINT MID$(M$,C,1)
360   NEXT
370
380 GCOL 4,1
390 PRINT TAB(0,16)"-----"
400 VDU 4
410 PRINT TAB(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 has the effect of keeping 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
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 " +/- "; 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 X2=(-B-SQR(DISCRIM))/(2*A)
330 PRINT "Real distinct roots
X=";X1;" and X=";X2
340 ENDPROC
350 @%=10:REPORT:PRINT

```

>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  
>

## 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
40 REM MODEL A
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 19,0,6,0,0,0
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** (e.g. line 170) results in a structure which is not easy to follow! It would be better to use "structures" such as procedures (see page 86).

```
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(1280)
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+1024-Y
170 IF X>1280 THEN 220
180 Y=1024
190 D=2
200 RETURN
210
220 Y=1024/1280-X
230 X=1280: D=4
240 RETURN
250
260 Y=Y-1280+X
270 IF Y<0 THEN 310
280 X=1280: D=3
290 RETURN
300
310 X=1280+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>1024 THEN 480
450 X=0: D=1
460 RETURN
470
480 X=Y-1024
490 Y=1028: 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. It runs on Model B only.

```

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;512;
90 J1%=0
100 FOR K%=500 TO 380 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 (lines 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 page 355).

```
10 REM SQRROOT
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 @%=10:PRINT:REPORT:PRINT
```

```

>RUN
What is your number ?34
Number          34.000
Root            5.831
Iterations      5.000
Time            0.070 seconds

What is your number ?125
Number          125.000
Root            11.180
Iterations      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 32 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
40 REM MODEL A
50 MODE 4
60 VDU 5
70 GCOL 0,1
80 VDU 19,1,1,0,0,0
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$="SINE WAVES ARE FAR MORE
INTERESTING . . . ."
180
190 FOR X=1 TO 39
200   MOVE X*1280/40,300*SIN(X/6)+512
210   PRINT MID$(A$,X,1)
220   NEXT
230
240 VDU 4
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
130 LDA #0A : 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.

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

**b** 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.

**c** Most keywords can be abbreviated using a full stop, eg **L .** for **LIST**, **SA .** for **SAVE**. See page 483 for a list of abbreviations.

# Giving the computer instructions – Part II

This section of the User Guide introduces many more of the 'keywords' which are used in the writing of programs on the BBC Microcomputer.

We start with some miscellaneous, useful keywords and then go on to look at some of the graphics and colour statements. It is easy to explore the effects which these produce without any systematic knowledge of BASIC. You should experiment with each statement not only by typing in the commands or test programs which are given but by changing many of the values of the various variables to see what effects are produced. In this way you will become familiar with the use of each keyword and begin to see how it can be used in your own programs. For further information see the reference chapter on BASIC keywords.

# 7 AUTO, DELETE, REM, RENUMBER

BASIC provides a number of facilities to help the user to enter programs into the computer and to 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 “offer” the line numbers automatically to the user. As an option you can tell the computer to start offering 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 on page 211.

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 quite often need to be able to delete a large chunk of it. The keyword **DELETE** is followed by two numbers which give the first and last lines that you wish to remove. Thus

```
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 **REMs** at the start of a program to give general information and then put a **REM** at major points further down the program. See pages 232 or 264 for examples.

Once you have entered a program you will very often find that the line numbers are no longer in a neat 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. Certainly, when you have finished a program it is a good idea to **RENUMBER** it so that it looks tidy. If you have a program in the computer try

**RENUMBER**

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

**RENUMBER 900,100**

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)  
PRINT "HELLO"
```

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

```
CLS : PRINT "HELLO"
```

You can put as many statements on a line as you like so 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 against that you will notice that it becomes much more difficult to follow the program when you list it (see page 98).



# 8 Introducing graphics

## Modes, Colours, Graphics and Windows

The Model A BBC computer can display text and graphics in four different modes; the Model B can show eight different modes. Only one mode can be used at a time. The Teletext Mode (MODE 7) differs from all the other modes in many ways and a whole chapter has been devoted to that mode (see page 150). In particular it is not easy to draw lines or triangles in MODE 7 and the colour of the text is changed in a special way. Finally some characters are displayed on the screen differently in this mode – for example the character `␣` is displayed as `←`.

On the Model A there are two modes in which graphics can be used (MODE 4 and MODE 5). With a Model B five graphics modes are available. The description which follows will assume that you are in MODE 5. To reach it, simply type **MODE 5** and press **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 with 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
```

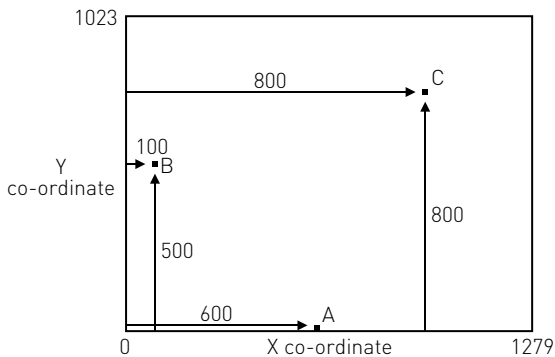
The same four colours (black, red, yellow and white) may be selected for the background with the commands

<b>COLOUR 128</b>	(128+0)	black
<b>COLOUR 129</b>	(128+1)	red
<b>COLOUR 130</b>	(128+2)	yellow
<b>COLOUR 131</b>	(128+3)	white

The **COLOUR** statement 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 – **GCOLOR**, which stands for Graphics COLOUR.

## Graphics

For the graphics: when drawing lines and triangles, positions on the screen are given with two numbers (the x and y co-ordinates).



The point A has co-ordinates 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 800,800. 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 1000,1000 type

**MOVE 1000,0**  
**DRAW 1000,1000**

**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 e.g.

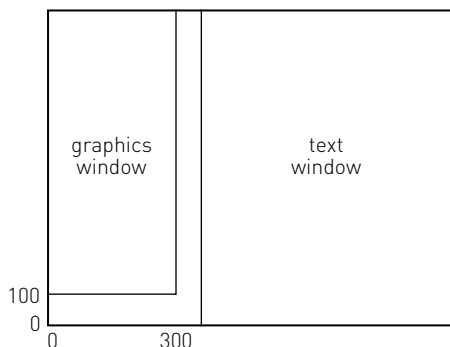
```
GCOL 0,0  black lines
GCOL 0,1  red lines
GCOL 0,2  yellow lines
GCOL 0,3  white lines
```

We'll consider what happens when the first number is not zero later on (page 167).

As with the text colours, one 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 can be quite clear which is which. We will then return to the **GCOL** statement.

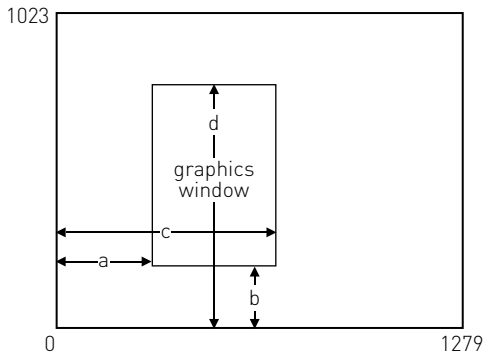
### Windows

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 (**MODES 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:



#### a 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: the comma after 24 and the semi colon after all the other values. The reason for this punctuation is given on page 386. So for our actual graphics window we would put

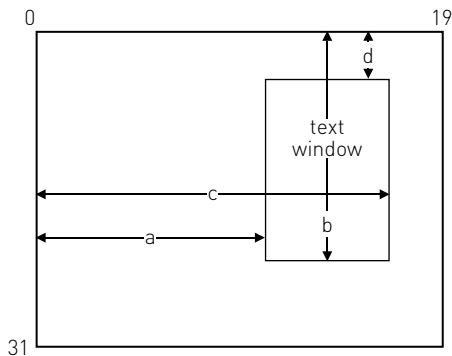
**VDU 24, 0;100;300;1000;**

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.

## **b 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 setup the window as follows

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

Note: the 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 28,5,31,19,0
```

To prove that we 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 (which support 80 characters to the line)	0 to 79
MODES 1, 4, 6 and 7 (which support 40 characters to the line)	0 to 39
MODES 2 and 5 (which support 20 characters to the 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** in the middle of what follows 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**. If you want to clear the graphics **only**, type **CLG**. (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 it has not done so visibly.

Windows may overlap – in fact when you change **MODE** both the text and graphic 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, e.g, in the middle of a program, use **VDU 26**.

**VDU 5** enables text to be drawn at any position inside a graphics window – see pages 49, 74 and 379.

### Changing the colours of text and graphics

Now back to text and graphic 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 4 at a time) by using **VDU 19**, as we saw on page 10. See page 382 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 7** 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 section 5.

# 9 More on variables

In an earlier section 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 rather odd – 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’, make it increase 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" e.g. 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 on page 24.

```
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 1 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 10 parts. Whilst you can add, subtract, multiply and divide using

numeric variables the only similar operation that can be carried out on a 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-emphasized. Later sections develop the 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 1982 can also be written as

MCMLXXXII	in Roman numerals
or 1982	in decimal Arabic numerals
or 7BE	in hexadecimal Arabic
or 11110111110	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
1001010	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 on page 490.

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**. So **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 section 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

170 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 (1.7×10<sup>38</sup>)

and 0.000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 (1.2×10<sup>-39</sup>), 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 9 figure accuracy, nonetheless this is quite accurate enough for most purposes.

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 to the nearest penny in £2M. Arithmetic calculations with integer variables are significantly faster than with real variables. (See section 32 for other suggestions for speeding up programs).

The two integer operators **MOD** and **DIV** are described on page 130.

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 WELCOME cassette uses the variable M%, to inform each of the individual programs whether or not the user's cassette recorder has got automatic motor control.

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\$
names	SIZE%	SIZE	SIZE\$
maximum size	2,147,483,647	$1.7 \times 10^{38}$	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 section 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$="HELLO"
30 PRINT X,X/2,X/4
40 PRINT A$,A$,A$
```

>RUN

HELLO              8                      4                      2

HELLO              HELLO              HELLO

← field →

width

### Field widths in different screen modes

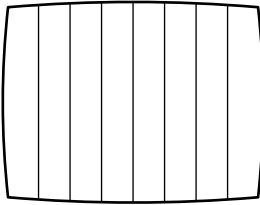
As we said above, the width of each 'field' is automatically set to ten characters when the computer is switched on.

Since the computer can operate in different screen modes, displaying either 20 or 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 re-running the program above.

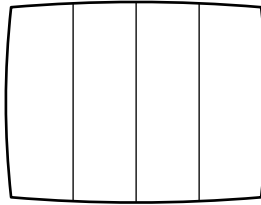
**5 MODE 5**

or, if you have a MODEL B machine

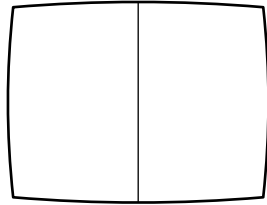
**5 MODE 0**



80 character modes  
(MODES 0 and 3)



40 character modes  
(MODES 1, 4, 6 and 7)



20 character modes  
(MODES 2 and 5)

Note: the widths of the fields can be altered by the use of a special command @% (see page 70).

So 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$="HELLO"
30 PRINT A$; X; A$; X; X
>RUN
HELL08HELL088
```

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 pairs of inverted commas exactly as they have been typed in, providing 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 quite 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 9 characters wide, and that numbers will be printed with a fixed number of decimal places – in this case, to 2 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

<b>&amp;</b>	<b>2</b>	<b>02</b>	<b>09</b>
(Means	Format	2 decimal	field width
Hexadecimal	number 2 i.e.	places	of 9
numbers	fixed number		characters
follow)	of decimal places		



`@%=&20309` would give Format 2, 3 decimal places and field width of 9 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 4 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

```

The `&` tells the computer that the numbers which follow are 'hexadecimal' numbers – that is, numbers based not on 10s but on 16s. For the sake of completeness, 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

A few points:

- 1 The maximum number of significant figures is 9
- 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 normal value (Format 0 and field width 10), set `@%=10`

## TAB(X)

As well as controlling the print layout by using the comma and semi-colon one 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 fifteen 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 bracket 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**

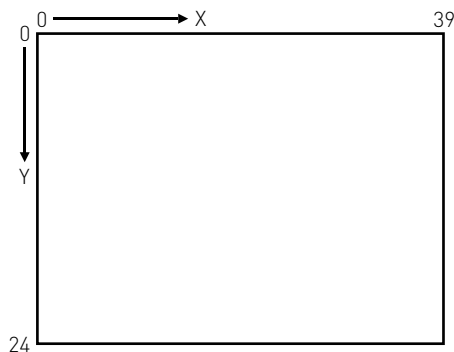
If you are beyond the place that you tell the computer to **TAB** to, for example in position 15 with a request to **TAB(10)**, then the computer moves to the next line and then tabs ten spaces.

Type in this replacement line

```
40 PRINT 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 co-ordinates are



This program counts to 1000, printing as it goes

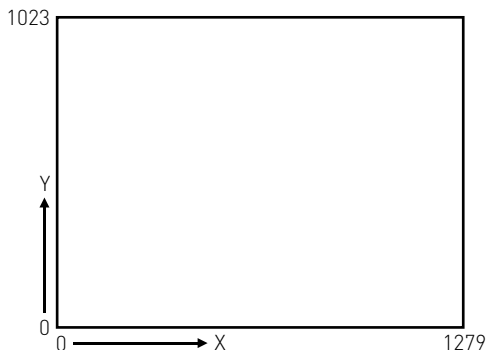
```
5  CLS
10 Q=1
20 REPEAT
30 PRINT TAB(18,5);Q
40 Q=Q+1
50 UNTIL Q=1000
```

The two numbers in the brackets after **TAB** represent the X and Y text co-ordinates where printing should start (see also the program on page 132).

### 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 **VDU 5** 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 thus



in all modes except MODE 7.

A few sums will show that 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 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 computer 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 `rôl e` with the accent correctly placed.

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

Once in `VDU 5` mode the screen will not scroll up if 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 `VDU 5` mode you can only print things in the graphics window and not in the text window, and colour is selected with the `GCOL` statement. `VDU 5` 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" amongst which are

code	effect
8	move cursor left
9	move cursor right
10	move cursor down
11	move cursor up

These code numbers can be used either with the `VDU` command e.g. to move left four spaces, use either

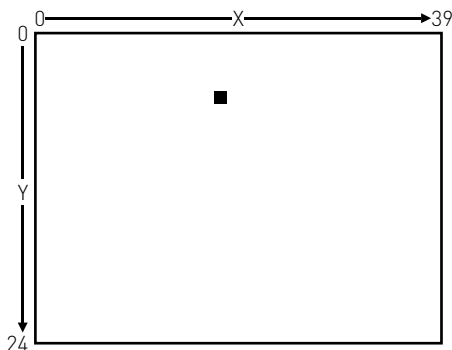
```
VDU 8,8,8,8
```

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 6 lines down. The cursor could be moved directly there with the statement

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

Note that the opening bracket 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. It can be turned off with the statement

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

and can be turned back on with the statement

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

or by changing screen mode with the **MODE** statement.

Note: In version 0.1 of the operating system the statement

```
VDU 23;8202;0;0;0;
```

must be used to turn the cursor off.

# 11 Input

The previous section showed how to get information out of the computer and on to the screen. This section 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 section 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 (e.g. **NAME\$**) on the input line.



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

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

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 ",NAMES$  
30 PRINT "HELLO ";NAMES$;" 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 **NAMES\$**. 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 " NAMES$  
30 PRINT "HELLO ";NAMES$;" 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 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 whereas 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 whereas

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

will wait for only 2 seconds (200 hundredths of a second). If no key is pressed within 2 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 a person to press any key when they are 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 MODE 4
```

```

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 one 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 MODE 4
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 computer 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 page 142). 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 on page 131.

## 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: rather like throwing a dice.

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

and this program draws random triangles in random colours

```
10 MODE 5
20 PLOT 85,RND(1200),RND(1000)
30 GCOL 0,RND(3)
40 GOTO 20
```

Sometimes it is useful to be able to re-set 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 brackets after the **RND** must be a negative number. Thus **X=RND(-8)** will ensure that the number sequence resulting from **RND(6)** is repeatable.

# Structure within BASIC

When building a house there are a number of options: one can chop trees down and tie them together to form a hut, one could use straw and mud if one had the skill to deal with those materials or one can use 'building blocks' like windows, doors and bricks. Whichever materials one selects the planning and construction is made much easier if one has a clear understanding of the available structures.

The same type of argument applies when writing a computer program to solve a particular problem. A number of program structures are available and it is well worth while learning what each can do. If you are attempting a trivial job (making an orange box or adding up a set of numbers) then little or no planning is needed. On the other hand, building an extension to your house or writing a computer program to play a game does require planning if it is not going to 'crash' without warning. This planning will involve the use of a number of 'structures'.

Here are the main structures available on the BBC computer.

```
REPEAT...UNTIL
FOR...NEXT
IF...THEN...ELSE
PROCEDURES
FUNCTIONS
GOSUB
ON...GOTO
ON...GOSUB
```



# 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 section explains **REPEAT . . . UNTIL** loops and the next chapter 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 of its own accord – 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 neatly together. See page 117 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 around, 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 around 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 6
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 5
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 7

and then re-listing the program.

>LISTO 7

>

>LIST

```
10 FOR ROW = 1 TO 6
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 5
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 10 stars

Lines 50 to 70 print out 20 equal signs

and Lines 10 and 90 ensure that the above are repeated 6 times.

Lines 100 to 150 print out 5 rows of 30 equal signs.

(With apologies to the USA for modifying their flag!)

### 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 on page 290. 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. Many 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 **LISTO0** 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 well 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 **LIST 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 around 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 on page 88 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 value of H, F and G at the time. As a result, the same program can behave in quite different ways in different circumstances.

### Multiple statement lines

It was explained earlier (page 54) 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

On page 89 the concept of **TRUE** and **FALSE** was introduced. A variable can have a numeric value (e.g. 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 and 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 from **-1** to **+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 sometimes in trying to fathom out what on earth the computer thinks it is doing!

# 17 Procedures

The BBC computer 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 MODE 7
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 @%=10

```

The program above shows how named procedures and functions can be used. The main 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 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 so 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 VDU 29,640;512;
60
70 FOR X=510 TO 4 STEP -7
80     GCOL 0,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" 510. **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 PROC SWITCH(A,B,C$)
```

```
1040 ENDPROC
```

it could not be called with a statement like

```
150 PROC SWITCH(X,Y)
```

but this would be acceptable

```
150 PROC SWITCH(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 variable 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 print-out 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 pretty pointless in its present form for several reasons – not least 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 to 50) and secondly, and quite separately, 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).

# 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
```

```
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 section, 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 beyond the **END** statement where it will not be executed except by being 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 quite straightforward – but in practice while the main aim is to write the letter there are often quite a few 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 those writing BASIC programs to break their programs up into recognizable 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 GOTO 460

```

```

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 `GOSUB` statement.

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; " Celsius"
  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
```

Certainly 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 on page 32 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 sheet:

```
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 8 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 have their place. 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 section.

# 21 Yet more on variables

## Arrays

Quite 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 5th entry or the fourth entry. The computer doesn't have any way of knowing that **N5\$** is the 5th entry. Using arrays, though, we can pick out the 5th 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 – e.g.

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

this 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 5 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.

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 5 letters

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

or print out every one 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 section 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 year 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 **DIM**ension **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 – e.g. **DIM Y(2000)**. You can also have “integer numeric arrays” like **DIM J%(100)**.

As usual on the BBC computer 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 Y(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 section 22 which deals with the keywords **READ**, **DATA** and **RESTORE**.

If you use **READ** and **DATA** to fill the above 5x3 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 – full of 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 5 entries in one dimension and 3 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 for 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 1, so an array declared with

```
DIM V(3)
```

has, in fact, got 4 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 section we set up a string array with the statement **DIM N\$(5)**.

This contains 6 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 go round 6 times. Each time it goes around it **READs** 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 around the loop.

```
>55PRINT 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 quite a neat way of getting to the (say) sixth element of a list but there is another way 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$="B" THEN RESTORE 170 ELSE
RESTORE 340
60   INPUT "HOW MANY POUNDS CAN YOU
AFFORD ",P
80   PRINT "YOU CAN AFFORD THESE
THEN:"
90   FOR X=1 TO 15
100      READ NAMES$
110      READ PRICE
120      IF PRICE<P THEN PRINT
PRICE,TAB(15);NAMES$
130      NEXT X
140      PRINT
150      UNTIL FALSE
160
170 REM BRITISH LEYLAND CARS
180 DATA MINI 1000 CITY, 2898
190 DATA METRO HLE, 4198
200 DATA DOLOMITE 1300,4351
210 DATA SPITFIRE 1500, 4696
220 DATA TRIUMPH ACCLAIM HLS, 4988
230 DATA ALLEGRO 31.3 HLS, 5095
240 DATA DOLOMITE 1500HL, 5225
250 DATA PRINCESS 2 HL, 5899
260 DATA DOLOMITE SPRINT, 7118
270 DATA TR7 FIXEDHEAD, 7258
280 DATA ROVER 2300, 7450
290 DATA DAIMLER SOVEREIGN 4.2,
16259
300 DATA JAGUAR XJ6 4.2,16279
310 DATA DAIMLER VANDEN PLAS 4.2,
21419
320 DATA DAIMLER LIMOUSINE, 26998
330
340 REM LOTUS CARS
350 DATA ESPRIT S3, 13513
360 DATA ECLAT SERIES 2.2, 14857

```

```
370 DATA TURBO ESPRIT, 16982
380 DATA ELITE SERIES 2.2, 17206
```

>RUN

```
DO YOU PREFER BL OR LOTUS CARS? B
HOW MANY POUNDS CAN YOU AFFORD ?10000
```

YOU CAN AFFORD THESE THEN:

2898	MINI 1000 CITY
4198	METRO HLE
4351	DOLOMITE 1300
4696	SPITFIRE 1500
4988	TRIUMPH ACCLAIM HLS
5095	ALLEGRO 31.5 HLS
5225	DOLOMITE 1500HL
5899	PRINCESS 2 HL
7118	DOLOMITE SPRINT
7258	TR7 FIXEDHEAD
7450	ROVER 2300

```
DO YOU PREFER BL OR LOTUS CARS? L
HOW MANY POUNDS CAN YOU AFFORD ?1700
```

YOU CAN AFFORD THESE THEN:

Out of DATA at Line 100

You will notice that line 50 uses the **RESTORE** statement to set the data 'pointer' to either line 170 where BL data is stored or to line 340 where Lotus data is stored. This ensures that data is read from the correct list.

Lines 90 to 130 attempt to read off 15 sets of data from the data lists but fail when Lotus data is selected as only 4 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 section 27 which deals with error handling.

# 23 Integer handling

Two special arithmetical functions are provided which produce integer (i.e., 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

$$11/4 = 2.75 \text{ or } 2\frac{3}{4}$$

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

$$11 \text{ DIV } 4 = 2$$

(i.e., 4 goes into 11 two times) and

$$11 \text{ MOD } 4 = 3$$

(i.e., 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=X 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
100 UNTIL FALSE

```

The clock is improved if you type **VDU 23;8202;0;0;0;** which switches off the flashing cursor (see page 77).

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 MODE 7
30 PROCOFF
40 PROCgetdatetime
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 12
1982"
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 month$
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;8202;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 TRUE 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 computer 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 3 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$="NOTWITHSTANDING"  
20 B$=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 going on for 9 letters”. As a result of this other 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.

3

6

0

```

10 A=45:B=30
20 A$=STR$(A)
30 B$=STR$(B)
40 PRINT A+6
50 PRINT A$+B$

```

>RUN

75

4530

>

i.e. line 40 treats A and B as numbers; and line 50 as string characters.

Note that **STR\$** is effected by the special variable @% if this has been set (see page 70).

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 imbedded 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(10,20); "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 CARNIVEROUS
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 unusual 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 some 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 inverted commas

**\*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 press the **RETURN** key when you set the key up. In fact you enter this to set up key f0.

**\*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 that any **\*KEY** definition must be the last statement on a line because since the computer finds a **\*** at the start of a statement it passes the whole of 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. This feature is implemented from release 1.0 of the operating system. Pressing **BREAK** and **SHIFT** together will be used in future to



“auto-boot” the machine so that the computer loads and runs a program without any further instructions. Note that “auto-boot” will not work on the cassette filing system.





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** 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 pages 422 and 423).

**COPY** 11

 12

 13

 14

 15

# 26 Operator precedence

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
$7 \text{ DIV } 4$	integer division
$7 \text{ MOD } 4$	integer remainder
$3^4$	raise to a power

These operators are referred to as binary operators since they require two operands (i.e., 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 precedence. 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
	<b>NOT</b>
	functions
	brackets ( )
	indirection operators (see page 409)

Group 2	<b>^</b>	raise to the power
Group 3	<b>*</b>	multiplication
	<b>/</b>	division
	<b>DIV</b>	integer division
	<b>MOD</b>	integer remainder
Group 4	<b>+</b>	addition
	<b>-</b>	subtraction
Group 5	<b>=</b>	equal to
	<b>&lt;&gt;</b>	not equal to
	<b>&lt;</b>	less than
	<b>&gt;</b>	greater than
	<b>&lt;=</b>	less than or equal to
	<b>&gt;=</b>	greater than or equal to
Group 6	<b>AND</b>	logical and bitwise <b>AND</b>
Group 7	<b>OR</b>	logical and bitwise <b>OR</b>
	<b>EOR</b>	logical and bitwise Exclusive <b>OR</b>

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. e.g. **IF NOT (X=5) THEN . . .** Clearly this example could be written **IF X <> 5 THEN** but the operator **NOT** is often essential when using functions e.g. **IF NOT FNVALID THEN . . .**

Functions include all the predefined functions such as **SQR**, **SIN**, **ASC** etc and user defined functions like **FNVALID**.

Brackets can be used to ensure that everything within the brackets is evaluated before any other calculations take place. Indirection operators are described in section 39 on page 409.

### Group 2

Raise to the power e.g.

$$3^2=9$$

$$3^3=27$$

### **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 e.g.

**IF X>10 AND Y=6 THEN ...**

For further details see the entry on page 205.

### **Group 7**

Logical **OR** is also used with multiple conditions e.g.

**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 page 250 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
```

**D i v i s i o n   b y   z e r o**

If you are sitting at the keyboard playing with the machine this is quite acceptable – 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 erroneous situation that might arise.

The major tool in error-handling is the statement

```
ON ERROR GOTO 5000
```

(the 5000 is for 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 an error handling routine in 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.

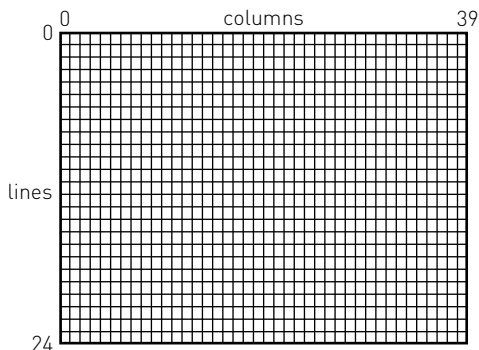
It is usually easy, but tedious, to anticipate all the probable errors but careful planning is needed if the error handling is to be effective. In particular you should be aware that when an error occurs you cannot return into a **FOR . . . NEXT** or **REPEAT . . . UNTIL** loop or into a procedure or 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.

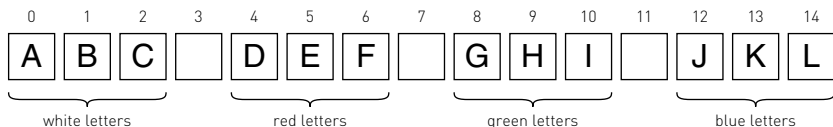
MODE 7 uses the standard Teletext control codes to change colours rather than the BBC BASIC **COLOUR** and **DRAW** statements. It cannot be over emphasized that MODE 7 requires different codes and statements from those available in modes 0 to 6. The MODE 7 display consists of 25 lines of 40 characters. 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 us 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) columns





Let us 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 **A B C** and the letters **D E F**. That space is in fact occupied by an invisible "control code". The control code 'appears' on the screen as a space 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 alpha-numerics (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, and 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 thus

```
10 MODE 7
20 PRINT "WHITE";CHR$(131);"YELLOW";
CHR$(135);"AND BACK TO WHITE"
```

### To make things 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 emphasized 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 so we can see what other effects are available.

To change the background colour requires 3 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 have come to appreciate it can be quite 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 table overleaf.

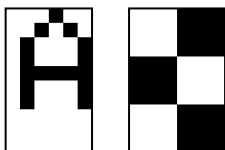
f0	128	No effect	
f1	129	Alphanumeric	red
f2	130	"	green
f3	131	"	yellow
f4	132	"	blue
f5	133	"	magenta
f6	134	"	cyan
f7	135	"	white
f8	136	"	flash on
f9	137	"	flash off

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 page 439 for more details.

Once you have a Teletext control code on the screen you can use the editing keys (e.g. **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 certainly less easy to use than in other modes but with a little patience some very good effects can be achieved. Another 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.



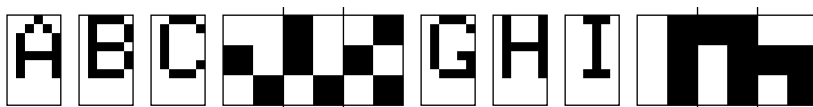
If you want to use those graphic shapes instead of lower case letters then you must precede them with one of the following control codes:

- 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



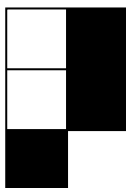
(The full list of graphic shapes are given on pages 488 and 489).

### Graphics codes

It is possible to calculate the code for any particular graphics shape in the following way. Each of the six cells contributes a certain number to the total code.

1	2
4	8
16	64

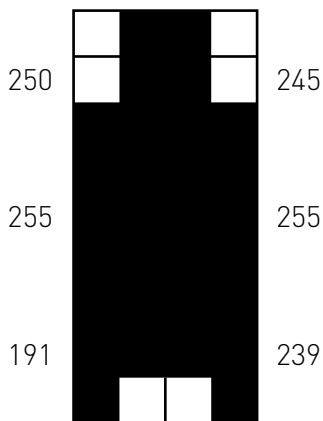
and *in addition* you should add in  $32 + 128$  (i.e. 160). For example the ASCII code for



is  $2 + 8 + 16 + 32 + 128 = 186$ .

## Making a large shape

Elsewhere in the guide it shows how to use user-defined characters to draw a space ship (see page 172). 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\$( )** so long as care is taken to get each in the correct place, e.g.

```
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 control codes is given on pages 486 to 489.

### 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 6 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
```

Next a row of teletext control codes must be written down the left hand side of the screen to turn 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 23
240 VDU 10,13,&97
250 NEXT
260 ENDPROC
```

The main program follows – in this case to plot a “sin curve”:

```
50 FOR X=0 TO 77 STEP 0.25
60 PROC PLOT(X,35+35*SIN(X/10))
70 NEXT
80 END
```

and lastly here is the procedure to plot the point

```
300 DEF PROC PLOT(X%,Y%)
310 LOCAL C%,A%
330 VDU 31,X% DIV2+1, 24-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 co-ordinates 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 the example on page 432 which explains this OBYTE 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 graphics 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%>77 OR Y%<0 OR Y%>74
THEN ENDPROC
```

and secondly we could remember the position of the cursor before we entered the procedure and restore the cursor to that position at the end of the procedure. See page 432 for a similar routine.



Notice that no reference need be made to actual memory co-ordinates and this is VITAL if your programs are to work via the 'Tube'. You may not think it important now but you will find that it is sensible to write programs using the machine code calls provided and not to get in 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** which selects a particular graphics mode  
**GCOL** which selects the colour and drawing 'style' of any graphics (except in **MODES** 3, 6 or 7)  
**DRAW** which draws lines (except **MODES** 3, 6 or 7)  
**MOVE** which moves the graphics cursor (except **MODES** 3, 6 or 7)  
**PLOT** which draws lines, dotted lines, points and colours in triangles (except **MODES** 3, 6 or 7)  
**VDU** a versatile command which produces a wide variety of effects.

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 (Model B only) with very high resolution and requires 20k of memory to "map" the screen  
**MODE 1** uses four colours (Model B only) with high resolution and requires 20K of memory  
**MODE 2** uses 16 colours (Model B only) with medium resolution graphics – 20k  
**MODE 3** text only – 16k (Model B only)  
**MODE 4** two colours and high resolution graphics – 10k  
**MODE 5** four colours and medium resolution graphics – 10k  
**MODE 6** text only – 8k

**MODE 7** teletext (which is the subject of a separate section of this book) – 1k

In MODES 0, 1, 2, 4 & 5 the screen is divided up into imaginary rectangles, like a piece of graph paper. In mode 0 there are  $640 \times 256$  squares; in MODES 1 and 4 there are  $320 \times 256$  and in MODES 2 and 5 there are  $160 \times 256$  (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.

### How to draw lines

On page 495 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

**a** 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). First set up a screen mode which can support graphics.

```
100 MODE 5
110 MOVE 400,400
```

**b** Draw a line horizontally to a 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
```

**c** 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 box

The normal foreground 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 4 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 MODE 5.

### **How to fill in with colour**

**PL0T 85,X,Y** (see page 319) 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 PL0T 85,800,800
```

Run this program.

This will fill in one half of the square

Now add:

```
170 PL0T 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 thus

Text foreground colour – white

Text background colour – black

Graphic foreground colour – white

Graphic background colour – black

The number of colours that can appear on the screen together 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 you say

```
COLOUR 0 to give black text
COLOUR 1 to give red text
COLOUR 2 to give yellow text
COLOUR 3 for 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
GCOL 0,3 white
```

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 on red. Graphics will still appear in white. To change text colours in the middle of a program merely 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 page 262). The second number refers to the logical colour which is to be used for graphics from now on. 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, but you can make one of those 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 any of the "logical colours" quite 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 e.g.

```
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

```
VDU 19,0,5,0,0,0
```

changes logical background colour 0 to actual colour 5, which is magenta. (Note. The zeros 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 a couple of other examples

```
VDU 19,1,2,0,0,0  logical colour 1 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 with **COLOUR 0**, or as the graphic foreground colour with **GCOL 0,0**. The table opposite shows how things are set up each time you change mode.

Foreground colour		Background colour	
Logical	Actual colour	Logical no	Actual colour
<b>Modes 0,3,4,6</b>			
0	Black (0)	128	Black (0)
1	White (7)	129	White (7)
<b>Modes 1,5</b>			
0	Black (0)	128	Black (0)
1	Red (1)	129	Red (1)
2	Yellow (3)	130	Yellow (3)
3	White (7)	131	White (7)
<b>Mode 2</b>			
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	Fl. Black/White (8)	136	Fl. Black/White (8)
9	Fl. Red/Cyan (9)	137	Fl. Red/Cyan (9)
10	Fl. Green/Magenta (10)	138	Fl. Green/Magenta (10)
11	Fl. Yellow/Blue (11)	139	Fl. Yellow/Blue (11)
12	Fl. Blue/Yellow (12)	140	Fl. Blue/Yellow (12)
13	Fl. Magenta/Green (13)	141	Fl. Magenta/Green (13)
14	Fl. Cyan/Red (14)	142	Fl. Cyan/Red (14)
15	Fl. White/Black (15)	143	Fl. White/Black (15)

You will have noticed 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 modes 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 colours of black, red, yellow and white the inverse colours would be white, yellow, red and black.

In **MODE 2**, the 16 colour effect 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 pages 205, 316 and 250 for a description of AND, OR and Exclusive-OR 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), i.e. 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 the production of sophisticated animation effects.

### **How to plot a point on the screen**

The **PLOT** command can also be used to plot points.

Type **MODE 4** (which is a 2 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 two points has gone. This is because **PLOT 70** prints the 'inverse' colour at the given point. This is particularly useful when 'animating', say, a point (see below). In **MODE 4** the 'inverse' of logical colour 1 is logical colour 0.

Suppose we want, say, 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 (i.e. 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 (i.e. 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 4 dots which move on the screen on a red background.

#### **a Setting up mode and colours**

```
10 MODE 4
20 VDU 19,1,3,0,0,0
30 VDU 19,0,1,0,0,0
```

**b** 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 tuck a procedure like this away at the end of a program, so we'll give it a high line number. X and Y are called 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.

c 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

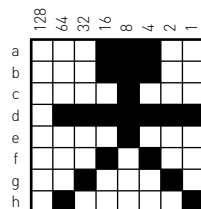
```

### 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 8 x 8 matrix of dots. The letter A, for example, has the code value 65 and 'a' has the value 97 (See page 490 for the other codes). However, certain code values have been left to be defined by the user. They include values 224 to 255 (See page 384 for more details). They can be defined by use of the **VDU 23** command.

### How to make a character (e.g. a man)

Create the character by planning it on an 8 x 8 square grid.



Note the numbers along the top of the grid which start at the right and double for every column moved 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
```

```
40 FOR T = 1 TO 100: NEXT T
50 PRINT TAB(X,10);" "
60 NEXT X
```

By altering lines 30 and 50 so that the value in brackets is (X,X) he can be made to move diagonally on the screen.

make this character 240

this 241

this 242

This time, a space ship.  
Plan this by using several grids.

```

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;

```

```

35 X=20
40 FOR Y = 24 TO 1 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 **VDU 5** statement (see also page 75, 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 graphic “grid” and print text or user defined characters. To do this **MOVE X,Y** is used.

**VDU 4** undoes the effect of a **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 from 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 page 75).

<b>VDU 8</b>	backspace cursor one character
<b>VDU 9</b>	forward space cursor one character
<b>VDU 10</b>	move cursor down one line
<b>VDU 11</b>	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 round. 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 3 times: here it is done in

**PROCROCKET**

```

5  MODE 4
7  VDU 23;8202;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%=20
37 VDU 5
38 GCOL 4,1
40 FOR Y%=120 TO 1023 STEP 10
50 PROCROCKET: REM draw the rocket
60 FOR F=0 TO 200 : NEXT : REM wait a
while
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 the '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 60 is a short delay and 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 almost a virtue. 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 lines 60 and 90, and type this new procedure:

```

38 GCOL 0,1
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 lander 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. We include 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;8202;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 **PROClab els** – sets up the titles

Line 130 **PROCmoon** – draws moon's surface

Line 140 **PROCinitialise** – sets all the variables to initial values

Lines 170 – 230 are the main part of program – a **REPEAT UNTIL** loop

Line 180 – checks to see if any key has been pressed

Line 185 – flushes 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 fuel burning

Line 240 – Crash or good landing? – at less than 15 mph it's good.

Line 247 – flushes keyboard buffer again so you don't get a whole string of numbers printed when the program stops.

```
700 DEF PROClab els
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 burnttime=(TIME-now)/100
1120 now=TIME
1130 slower=(burnrate%/totalmass)*2*
EXP(burnrate%*burnttime/totalmass)
1140 height=height-speed*burnttime-
burnttime*burnttime/2*(gravity-slower)
1150 speed=speed+burnttime*(gravity-
slower)
1160 burnt=burnrate%*burnttime
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)(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     GCOL0,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 16500 lbs 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.

# 30 Sound

The BBC computer 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 explored. At its simplest the sound statement is followed by four numbers e.g.

**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.

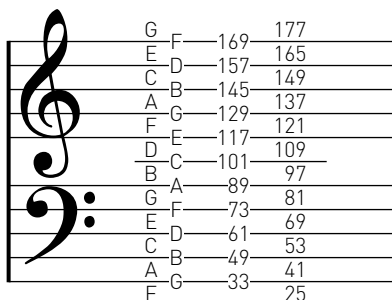
Middle C is produced when P is set at 53 and other notes are generated with the values of P given opposite.

As you can see the computer can produce notes spanning five full octaves. The values of P are also shown opposite for a stave in the 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  $\text{♩} = 60$ " will sound about right with the following settings for D.



Note	Octave number						
	0	1	2	3	4	5	6
B	1	49	97	145	193	241	
A #		45	93	141	189	237	
A		41	89	137	185	233	
G #		37	85	133	181	229	
G		33	81	129	177	225	
F #		29	77	125	173	221	
F		25	73	121	169	217	
E		21	69	117	165	213	
D #		17	65	113	161	209	
D		13	61	109	157	205	253
C #		9	57	105	153	201	249
C		5	53	101	149	197	245



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, PI2, PI3, PN1, PN2, PN3, AA, AD, AS, AR, ALA, ALD**

A brief description of each parameter follows

<b>Parameter</b>	<b>Range</b>	<b>Function</b>
N	1 to 4	Envelope number
T bits	0-6	1 to 127 length of each step in hundredth of a second
bit 7	0 or 1	0 = auto repeat pitch envelope, 1 = don't 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



Note that the pitch can take on a value between 0 and 255. If the pitch is greater than 255 (e.g, 257) then 256 will be repeatedly subtracted from it until it is in range.

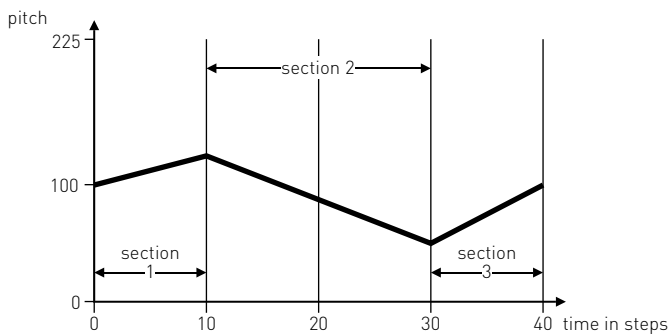
The amplitude has a range of 0 to 127 in the **ENVELOPE** statement whereas it had a range of 0 to -15 in the **SOUND** statement. The amplitude cannot be set outside the range 0 to 127.

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 of 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 3 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 i.e. 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.

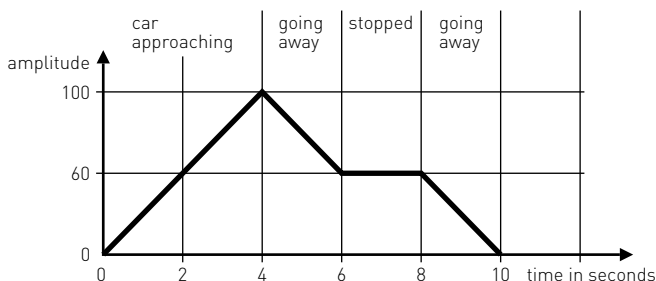
The third section has a pitch change of 8 units.

All three sections have only 1 step in each section.

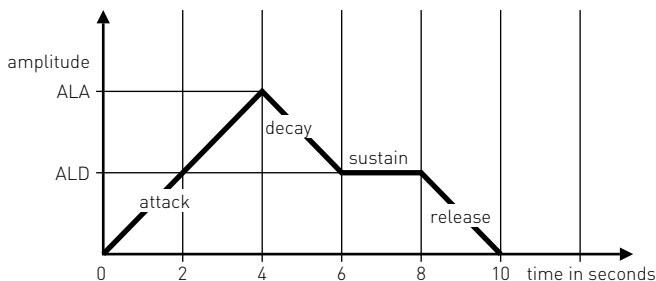
Now to explain the amplitude envelope.

### 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
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

In our example the attack phase takes 4 seconds and each step lasts 1/4 second 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 7 we will get there in 16 steps. So parameter AA = 7.

During the decay phase the amplitude must drop from 100 to 60 in 2 seconds. During 2 seconds there are 8 steps. So the amplitude drops 40 units (100-60) in 8 steps – so each step must reduce the amplitude by 5 units. Thus AD = -5. So far we have determined the following parameters of the amplitude envelope.

```
AA = 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 numerous ways of getting things 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 quite 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,
-2,120,120
```

## Note synchronization 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 = &H S F N**

Parameter	Range	Function
N	0 to 3	Channel number itself
F	0 or 1	Flush control
S	0 to 3	Synchronization control
H	0 or 1	Continuation control

**N** selects the channel number.

**F** If **F** is zero the sound will be placed in the 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 synchronize 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 event on the channel to continue if it is set to 1. In this case the amplitude and pitch 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 continued. 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 synchronized with 2 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** are given in the keyword section on pages 347 and 244.

# 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 Computer 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 network systems. Programs written to work on a cassette file system will usually work unmodified on a disc system. See page 400 for details of other file systems.

Firstly 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.
- 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 5 telephone lines to 5 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 tidy and you really have no idea how he looks after the files – nor does it matter, so long as he is efficient.

To return to our computer, suppose that we wish to create a file called "**DRINKS**" in which we list every drink we ever try. 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\$**

and then we could 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
```

```

RECORD then RETURN
What is the drink called?WHISKY
What is the drink called?PORT
What is the drink called?GIN
What is the drink called?BEER
What is the drink called?LAGER
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 or disc. The program to read the file back in is also quite 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*, if the tape is now played from a point just before the position where the data was recorded, the list will then appear on the screen.

```

>RUN
WHISKY
PORT
GIN
BEER
LAGER
CIDER
TOMATO JUICE
STOP

```



When using these programs on cassette you must be careful to position the tape (by "forward" and "rewind") so that the tape is in the right place. In the first program, where you record the data onto the tape you will have to find a piece of blank tape – preferably with the tape recorder counter set to 100 or 200 or 300 or some other round number. After you have finished the first program you have to rewind the tape to the start of the data file which was at, say, 200. It is just like trying to play a piece of music back that you have recorded on a cassette; you must rewind the cassette to the start of the piece. You may well know how difficult it is to find a particular record on a cassette and that is why it is so important to use the tape counter every time and to write down where each title is on the tape. Your index might look like

```
100 NAMES
200 DRINKS
300 SLIDES
400
```

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 page 397 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 specialist 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 from laboratory experiments.

EXT# and PTR# are mainly used with disc systems where

random access files are required. They cannot be used in a cassette environment.

**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 (e.g. `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`. i.e., say `NEXT` rather than `NEXT X`. This saves quite a lot of time.
- 3 `REPEAT . . . UNTIL` loops are much faster than `IF . . . THEN GOTO` loops.

4 Procedures are faster than **GOSUBs**, and it is faster to pass parameters to a procedure than to use global variables. i.e, 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 – i.e. 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 all 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.

# Reference section

# 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. 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 section 9 on page 62 first. The mathematical functions SIN, COS, TAN, etc are not explained for the absolute beginner – there just isn’t enough room to explain everything!

## **Examples**

This section gives a few one-line examples of the use 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.

## Known problems

Any known bugs are described here.

## Syntax

The syntax of each keyword's usage is given more by way of helpful explanation than for its rigorous accuracy. Purists will, rightly, complain at this travesty of Backus-Naur Form (BNF). Others may find the entries useful.

The following symbols are used as part of the syntax explanation:

**{ }** denotes 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 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 for example "4\*X+6"

**<string-const>** means a string enclosed in quotation marks, e.g. "GAVIN MOUNT"

**<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\$+"WOMBAT"**

**<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 pages 24, 62 and 120).

### **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.

# 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.

## Known problems

There is a known 'bug' in release 1.0. If **ABS** is used with the unary minus operator (**PRINT**  $-\text{ABS}(1)$ ) a 'Type mismatch' error message will be reported.

## 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 much prefer to work with whole numbers; for example

16h : 15m : 23s

There are four “Analogue to Digital” converters in the Model B 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 “joy-stick” 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 0V 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 large range – enabling the computer to measure things more accurately. In order to ensure that the BBC Microcomputer can cope with this situation we have specified a large range.

Thus instead of numbers in the range 0 to 4095 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 Model B microcomputer and the number in brackets 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 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) DIV 256**

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 brackets – e.g. **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

<b>X=ADVAL(-5)</b>	returns the number of free spaces in the SOUND channel 0 buffer
<b>X=ADVAL(-6)</b>	returns the number of free spaces in the SOUND channel 1 buffer
<b>X=ADVAL(-7)</b>	returns the number of free spaces in the SOUND channel 2 buffer
<b>X=ADVAL(-8)</b>	returns the number of free spaces in the SOUND channel 3 buffer
<b>X=ADVAL(-9)</b>	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.

e.g.

```
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 4 channels each of 12 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 results so that the function **ADVAL** returns very quickly. New samples are taken about every 10 milli-seconds. Thus with 4 channels selected results will be updated every 40 ms. See page 426 "Machine Operating System" 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” operator. A common jargon word for “bit by bit” is “bitwise” and it will be used in the description which follows.

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 the **IF...AND...THEN...** construction.

Bitwise **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 bit 1 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	0000	0000	0000	0000	0000	0000	0000	1110
7	0000	0000	0000	0000	0000	0000	0000	0111
6	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 integer bitwise **AND** between two items. Note that the logical and bitwise 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>**

**<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). Most 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 1-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**. 1 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(2.31))
```

## 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 line 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 largest 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 by 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 floppy discs. 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 page 188 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, page 188, 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, OPENOUT, OPENUP, CLOSE#, EXT#, PTR#, PRINT#, INPUT#, BPUT#, EOF#**

# BPUT# put a byte to file

## Purpose

To store a byte on the cassette or disc. See page 188 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. Large 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**, **OPENOUT**, **OPENUP**, **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 for those familiar with 6502 assembly language is given in section 43.

## 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 as necessary.
parameter type	



**Parameter types:**

- 0 - 8 bit byte (e.g. ?X)
- 4 - 32 bit integer variable (e.g. !X or X%)
- 5 - 40 bit floating point number (e.g. V)
- 128 - A string at a defined address (e.g. \$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 payslips. In turn that might then **CHAIN** a third program which would do a coin 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 could then **CHAIN** the main game – thus releasing a lot of the computer's memory.

**CHAIN** " " will chain the next program (on a tape). This will not work with other file 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. 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 on page 486.

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 least 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 in use previously, including string variables and arrays but excluding the “resident integer variables” @% and A% to Z% which are not affected in any way. See page 62 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 @%...Z%

## Syntax

```
CLEAR
```

## Associated keywords

None

# CLOSE# close file

## Purpose

To inform the computer that you have completely finished with a particular file (see page 188 for an explanation of “files”). The computer then transfers any data still in memory to cassette, disc or Econet as needed. See the chapter on file handling (page 188) 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, OPENOUT, OPENUP, EXT#, PTR#,  
PRINT#, INPUT#, BGET#, BPUT#, EOF#
```

# 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**, page 262, 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
```

# 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** on page 222 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 left of the current text area.

## 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 sixteen effects by using another command. Remember that the colours given above (black, red, yellow, white) will be available as soon as a four colour mode is selected – but you can then select others 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” 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 the chapter 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 brackets (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 automatically make available 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 5 countries and their 5 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 quite clumsy at handling information like this, as the demonstration program on the next page shows.

In the example program the **DATA** consists of lots of 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 **h e l l o** will just give 0.

There is no need to put each word in inverted commas unless leading spaces are important in the **DATA** words, for example

"        **f o u r   s p a c e s** " .

If you wish to have leading spaces then these words should be enclosed in inverted commas. Since a comma is used to separate items of **DATA**, if you want a comma in your **DATA**, you must enclose the **DATA** in inverted commas.

## 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 5 CITIES and the other to store the names of 5 countries. See page 120 for an explanation of arrays.

Line 40 sets up a FOR . . . NEXT loop that will go round 5 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 10 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 define

## 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.

The definition 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 = FN VAT**. 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 not



defined 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 FNVAT(g)=1.15*g
```

“g” is called a “formal parameter” for the function **FNVAT**. 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 ";  
235 PRINT FNVAT(P)
```

and in this case “P” is the “actual parameter” for the function **FNVAT**. Whatever value “P” has will be used inside the procedure wherever reference is made to the “formal parameter” “g”. This is very convenient since you can use any variable names that 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) 3 formal parameters then, when it is called, 3 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 a 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 FNVAT(P)
240 UNTIL P=0
250 END
1000 REM one Line numeric function
1010 DEF FNVAT(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 1 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 COLOUR 128+X
30 CLS
```

```

40 DRAW 1279,0
50 DRAW 1279,1023
60 DRAW 0,1023
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. Procedures and functions need not be defined before they are called. All procedures and functions must be placed in the program where they will not be executed e.g. after the **END** statement.

### Syntax

```

DEF FN|PROC<variable name>[(
<string-var>|<num-var>{,<string-var>|
<num-var>})]

```

### Associated keywords

ENDPROC, FN, PROC

# DEG degrees

## Purpose

This function converts angles which are expressed in radians into degrees. 1 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 one wanted to work with a set of information about the rooms in an hotel with 4 floors, each with 30 rooms, then an array of 4 by 30 entries can be created thus

```
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 £23.50

```
hotel(1,22)=23.50  
hotel(4,1)=145.00
```

In practice the statement **DIM hotel(4,30)** produces an array of 5 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. e.g.

```
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 bracket. 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 **DIM X(4)** would create an array of 5 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 brackets 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>})  
DIM <num-var> <numeric>
```

### Associated keywords

None

# **DIV** division of whole numbers

## **Purpose**

See page 299 which describes the word **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 converted 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 MODEs 0,1,2,4 and 5. The **DRAW** statement is followed by two numbers which are the X and Y co-ordinates 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** on page 319 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. See the pages describing the associated keywords and section 16 page 98 for more details.

## Examples

```
560 IF length > 0 THEN PRINT "O.K."  
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>]
```

## 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** on page 230 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

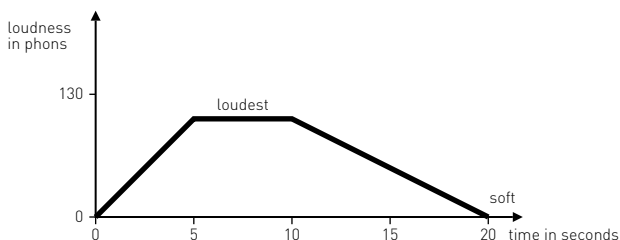
**DEF, FN, PROC, LOCAL**

# ENVELOPE

## Purpose

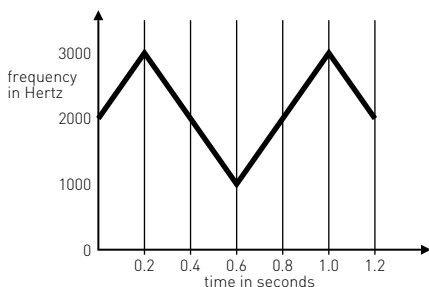
The envelope statement is used with the **SOUND** statement to control the volume and pitch of a sound while it is playing. All natural sounds change in volume (loudness or amplitude); for example, the sounds from a piano start off loudly and then fade away. An aircraft flying overhead starts off softly, gets louder and then fades away.

The variation of amplitude (loudness) for the aircraft, as it flew overhead, might well look 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 computer 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.

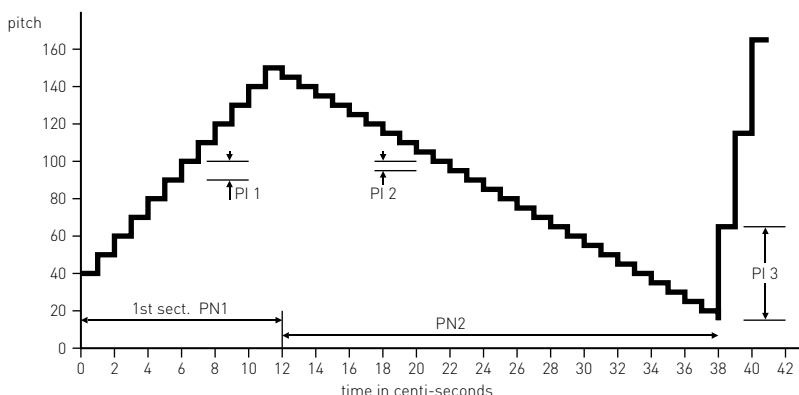
```
ENVELOPE N,T,PI1,PI2,PI3,PN1,PN2,  
PN3,AA,AD,AS,AR,ALA,ALD
```

Parameter	Range	Function
N	1 to 4	Envelope Number
T bits	0 to 127	Length of each step in hundredths of a second
bit 7	0 or 1	0=auto-repeat the pitch envelope 1=don't auto-repeat
PI 1	-128 to 127	Change of pitch per step in section 1
PI 2	-128 to 127	Change of pitch per step in section 2
PI 3	-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 of level at end of attack phase
ALD	0 to 126	Target of 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 centi-seconds 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** – i.e. using values of **T** greater than 127.

The six parameters **PI 1**, **PI 2**, **PI 3**, **PN1**, **PN2** and **PN3** determine the pitch envelope. The pitch envelope has 3 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** = 1 centi-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.

$$\text{time} = (\text{PN1} + \text{PN2} + \text{PN3}) * \text{T centi-seconds}$$

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 “fold over”, that is the value used is **MOD 256** of the value calculated.



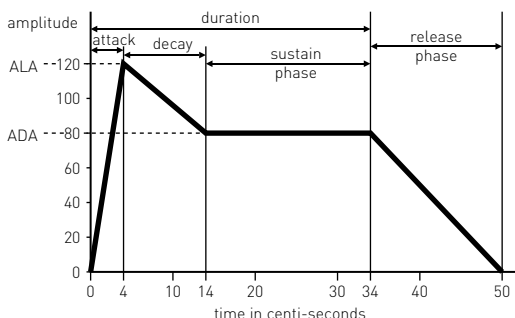
The six parameters **AA**, **AD**, **AS**, **AR**, **ALA** and **ALD** determine the amplitude envelope. Although the current internal sound generator has only 16 amplitude levels the software is upward compatible with a generator having 128 levels.

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, D, 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

$T = 1$  centi-second

$ALA = 120$

$ALD = 80$

AA = 30 (120 in 4 centi-seconds)  
AD = -4 (-40 in 10 centi-seconds)  
AS = 0  
AR = -5 (-80 in 16 centi-seconds)

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# end of file

## Purpose

This function is used to tell 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 page 192 for more information.

## Example

```
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, OPENOUT, OPENUP, 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 page 205, which explains 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 1, if they are the same it will be set to 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          0000 1100 0011 0000 1110 1011
Y          1011 1111 0000 1010 0010 1000
then
X EOR Y    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 bitwise exclusive-or between the two operands.

## Syntax

```
<num-var> = <numeric> EOR <numeric>
```

## 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 page 147 for more information.

## Example

```
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 GOTO, ON ERROR OFF, REPORT,
ERR
```

# ERR error number

## Purpose

If the computer finds an error that it cannot cope with, 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 usually 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 GOTO, 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".

This function is not common in other versions of BASIC so a few more specific examples are given of legal 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 only be used to evaluate functions (**SIN**, **COS**, **SQR** etc.) and 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>)
```

```
<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

The 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 finds out how large a particular file is. It only works with disc and network file 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 using the **OPENIN** or **OPENOUT** statements. See page 188 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**, **OPENOUT**, **OPENUP**, **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 a result the program will never print  $X \text{ EQUALS } 20$ .

```
100 X=12
110 REPEAT
120 PRINT "HELLO"
130 UNTIL X=20
```

would repeatedly print **HELLO** because  $X$  will never be anything else 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 one 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** on page 230 and sections 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 equals 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

```
FN<variable-name>[( <num-var> |  
<str-var>{,<num-var>|<str-var>} )]
```

## Associated keywords

**DEF, LOCAL, PROC, ENDPROC**

# 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

```

99 Hi there
87 Hi there
75 Hi there
63 Hi 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.

### Syntax

```

FOR <num-var> = <numeric> TO <numeric>
[STEP <numeric>]

```

### Associated keywords

**TO, STEP, NEXT**

# GCOL graphics colour

## 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.

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.

## Example

```
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 page 64 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 wants another go. 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 then ask the computer to print it.

## Example

```
1050 A$=GET$
```

## Syntax

```
<string-var> = GET$
```

## Associated keywords

**GET**, **INKEY**, **INKEY\$**

## Demonstration Program

```
2100 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**

Quite 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 one can separate out a small sub-section into a 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>
```

## **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$="hand moutheat leg arm
chestelbow"
120 FOR count=1 TO 10
125 REM pick a random number
130 R=RND(7)
140 REM and use it to pick a random
word
150 B$=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$;" alright?"
230 NEXT count
```

Now look at the same program using a subroutine and with the REMs (remarks) removed.

```
110 A$="hand moutheat leg arm
chestelbow"
120 FOR count=1 TO 10
150 GOSUB 810
170 PRINT "My ";B$;" hurts"
190 GOSUB 810
220 PRINT "Is your ";B$;" alright?"
230 NEXT count
240 END
810 B$=MID$(A$,5*RND(7)-4,5)
820 RETURN
```

The "7" in line 810 is there to select one of the 7 words in A\$. In line 810 both the "5"s 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 REMs left out and generally tidied up.

```
110 A$="hand moutheat leg arm  
chestelbow"  
120 FOR count=1 TO 10  
170 PRINT "My ";FNword;" hurts"  
220 PRINT "Is your ";FNword;"  
alright?"  
230 NEXT count  
240 END  
800 DEF FNword=MID$(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, if the program is renumbered using the **RENUMBER** command, the program will probably then branch to the wrong line.

Note that if the destination line number is to be calculated using a mathematical expression then that expression must be in brackets.

**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
```

```
100 GOTO (starts*55+14)
```

**Description**

A statement used to transfer control to a specified or calculated line number unconditionally.

**Syntax**

**GOTO** <numeric>

# **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 in a sensible way. 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 on page 500 gives an indication of their relative positions.

## **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!

**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>
```

or

```
IF <testable condition> THEN <line  
number>
```

## Associated keywords

**THEN, ELSE**

# **INKEY** input the number of the key pressed

## **Purpose**

This function waits for a specified time whilst 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** on page 207 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 “flush” the buffer of all characters by giving the command

**\*FX 15,1**

The number in brackets, 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 brackets e.g.

**INKEY (-27)** will enable you to test to see whether a particular key is pressed at that instant. The number in brackets 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 centi-seconds.

### 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
M	-102	<b>ESCAPE</b>	-113
N	-86	<b>TAB</b>	-97
O	-55	<b>CAPS LOCK</b>	-65
P	-56	<b>CTRL</b>	-2
Q	-17	<b>SHIFT LOCK</b>	-81
R	-52	<b>SHIFT</b>	-1
S	-82	<b>SPACE BAR</b>	-99
T	-36	<b>DELETE</b>	-90
U	-54	<b>COPY</b>	-106
V	-100	<b>RETURN</b>	-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 whilst 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 flush the buffer of all characters by giving the command

```
*FX 15,1
```

The number in brackets, 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 centi-seconds.

## 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 – for example “7”. 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. **INPUT** can be used to get in words as well.

```
205 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 the lot into **A\$**.

### **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**

Too complicated for a useful yet simple description.

### **Associated keywords**

**INPUT#**



# **INPUT#** put information into the computer from cassette or disc

## **Purpose**

It is possible to record data (numbers and words) on cassette or floppy 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 the section on file handling on page 188 for more information.

## **Example**

```
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, OPENOUT, OPENUP, 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$,B$)
```

put the position of B\$ in A\$ into X

```
180 Y=INSTR(A$,B$,Z)
```

start 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 bracket.

## Syntax

```
<num-var> = INSTR(<string>,<string>[,<numeric>])
```

### **Known problems**

There is a known “bug” in release 1.0. If the second string is longer than the first string, for example,

```
X=INSTR("A","KNOWN BUG")
```

then the function will appear to work but will corrupt the stack. *This bug is fatal if INSTR is used in procedures or functions.*

### **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-hand end of the source string. For example if

```
A$="CATASTROPHE"
```

then

```
PRINT LEFT$(A$,3)
```

would give CAT, namely the left 3 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 bracket.

## 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 by 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** on page 207 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 A=A+10
```

```
1000 IF length<12 THEN LET length=12
```

## Description

**LET** is an optional assignment statement.

Note: **LET** may not be used during the assignment of the psuedo-variables **LOMEM**, **HIMEM**, **PAGE**, **PTR#**, **TIME**

## Syntax

**LET <var> = <expression>**

# LIST

## Purpose

This command makes the computer list out 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 that, and to make the computer stop at the bottom of each page you can type **CTRL N** (whilst holding down the key marked **CTRL** press the *letter N*). Then type **LIST**. This is called "paging 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 the *letter O*).

Pressing **SHIFT** and **CTRL** together holds the listing until they are released.

Pressing **ESCAPE** stops the listing.

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 **LIST0** (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, LIST0**

# 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

- a after the line number
- b during **FOR . . . NEXT** loops
- c 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 **LISTO 6** would be selected. **LISTO 7** puts a space after the line number and double spaces for **FOR . . . NEXT** and **REPEAT . . . UNTIL** loops.

The most common options are **LISTO 0** and **LISTO 7**.

When editing programs using the cursor editing keys it is strongly advised that you use the **LISTO 0** 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 on **FOR . . . NEXT** loops; Bit 2 the double space on **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(REDOPotential)
```

## Description

A function returning the natural logarithm of its argument.

## Syntax

```
<num-var> = LN(<numeric>)
```

## Associated keywords

LOG, EXP

# LOAD

## Purpose

To load a program into the computer from cassette tape, floppy disc or the network, whichever is the current file system.

A WELCOME cassette containing some games is provided with the computer and these games can be loaded into the computer with the keyword **LOAD**. If you have problems loading a program then see page 34. The keyword **LOAD** is followed by the 'file name'. This file name may contain up to 10 characters and must be enclosed in quotation marks, e.g.

```
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 that it had in memory and also the values of all variables.

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.

**LOAD** does not *run* a program. It just loads it into memory. It clears all variables except A% to Z% and @%. It cannot be used in a program.

**LOAD ""** will load the next program found on a tape. This form does not work on disc or net or other file systems.

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 file 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** on page 230 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 program

```
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

$$Y=10^X$$

## Syntax

```
<num-var> = LOG(<numeric>)
```

## Associated keywords

LN, EXP

# LOMEM lowest memory location

## 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**.

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** on page 270 and the memory map on page 500 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
```

NB 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\$ middle string

## Purpose

To copy part of one string into another string. For example if

```
demo$="DOGMATIC"
```

then the middle part of **demo\$**, starting at position 4 and going on for 3 letters i.e.

```
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 on page 266 under the keyword **GOSUB** and another under the keyword **LEN** on page 285.

If the last number is omitted then the function returns with the rest of the string.

## Example

```
RESTofLINE$=MID$(main$,10)
```

## Description

A string function which returns a subsection 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 forwards 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=2
14.6  DIV 5.1=2
14    MOD 5=4
14.6  MOD 5.1=4
```

The second example (  $14.6 \text{ DIV } 5.1$  ) is really the same as the first. However.

```
14.6  DIV 4.9=3 and
14.6  MOD 4.9=2
```

are quite different. In effect the computer sees them as

```
14    DIV 4=3
14    MOD 4=2
```

### 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

$$A \text{ MOD } B = A - ((A \text{ DIV } B) * B)$$

### 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. On the Model A computer display modes 4, 5, 6 and 7 may be selected. On the Model B all display modes are available. Changing modes clears the screen.

Mode	Graphics	Colours	Text
0	640×256	2 colour display	80×32 text
1	320×256	4 colour display	40×32 text
2	160×256	16 colour display	20×32 text
3		2 colour text only	80×25 text
4	320×256	2 colour display	40×32 text
5	160×256	4 colour display	20×32 text
6		2 colour text only	40×25 text
7		Teletext display	40×25 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 modes 0 to 6 the character set can be changed by the user. See **VDU 23** on page 384.

You cannot change **MODE** inside a procedure or function.

## 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**.

### 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 **PLOT 4**.

## 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 statements 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>]
```

## 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 and tidy 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 800,920,100,1170,7300
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 1, 2 or 3 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 give 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 the page describing **GOSUB** for an explanation of subroutines.

**ON X GOSUB** provides a neat way of using different subroutines in different situations.

**Note:** 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. Unfortunately this facility upsets returning from functions and procedures in the first version of **BASIC**.

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 cope with, 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 **T o o b i g**.

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 piece 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 a thing 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>}
or
ON <num-var> GOSUB<numeric>{,<numeric>}
or
ON ERROR <statement>
or
ON ERROR OFF
```

### Associated keywords

GOTO, GOSUB



# **OPENIN** open file for input to computer (from cassette or disc)

## **Purpose**

To tell the computer that your program wishes to read data (words and numbers) from the cassette or disc. Reading data in from cassette or disc 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" (i.e. the cassette or disc) and to read the file in each time you want to update it. After you have corrected it you can then transfer it back to cassette or disc where it will be saved for future use. Further information about cassette, disc and network "files" is provided on page 188.

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 on cassette or disc under the name "cinemas". The "filename" is "cinemas".

In accepting this instruction the computer allocates a "channel" to this operation. It is as if it said "O.K. 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 (from the cassette) by using **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 network environment then if a file already exists with the correct name it will be opened for updating (reading or writing).

The function returns the channel number allocated by the computer's file 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 cassette 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!
```

# OPENUP open file for update

See page 513 for details.

# **OPENOUT** open file for output to cassette or disc

## **Purpose**

This opens a cassette or disc file for output. Before you can record data (rather than programs) onto a cassette you have to "open a file". More information about "files" is given on page 188.

**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 the network 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#
```

# 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 **0%** gives the memory location where the code is placed.

The **OPT** statement can only occur inside the square brackets which enclose a piece of assembly language. **OPT** is set to 3 every time the BASIC interpreter finds a **[**. Do not confuse it with **\*OPT** which is described on page 434.

## 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
```

# **OSCLI** command line interpreter

See page 514 for details.

**Description**

A statement which passes its string argument to the operating system command line interpreter.

**Syntax**

**OSCLI**<string>

**Associated keywords**

**STR\$**, **CHR\$**

# 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 “bitwise or”. See the keyword **AND** on page 205 for details of logical and bitwise operators.

## Example

```
75 IF X=6 OR date>20 THEN PRINT  
"Good"
```

## Description

An operator performing bitwise integer logical **OR** between two numerics.

## Syntax

```
<num-var> = <numeric> OR <numeric>
```

## Associated keywords

**AND, EOR, NOT**

# 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 co-ordinates 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 current graphics background colour
- 4        move to absolute position
- 5        draw line absolute in the current graphics foreground colour
- 6        draw line absolute in logical inverse colour
- 7        draw line absolute in 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 in inverting actions' – eg using **GCOLOR 4**
- 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 are reserved for the Graphics Extension ROM
- 64-71 as 0-7 but only a single point is plotted
- 72-79 as 0-7 but plot points left and right from the given coordinates until a non-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 plot points to the right from the given coordinates until the background colour is reached
- 96-255 reserved for future expansions.

See the vdu driver section of the User Guide on page 377 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 modes the logical inverse colour of logical colour 0 is logical colour 1.

In four colour modes the following apply

<b>logical</b>	<b>inverse</b>
0	3
1	2
2	1
3	0

In the sixteen 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 to the 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 **VDU 29** statement. See page 388 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 cursors together with the statement **VDU 5** 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 graphic 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 bracket.

## 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 co-ordinates. 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 behaves as if it is divided into vertical strips (or fields) which are (initially) 10 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.

`PRINT ~` followed by a numeric argument prints that argument in hexadecimal.

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
	,,,,,,,,,2.25

The printer can be turned on at any time by typing **CTRL B** or by the statement **VDU 2** 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 page 404 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 2 decimal places

**@%=131850** will give 3 decimal places

**@%=10** will return to the normal output format.

For a detailed understanding of the format it is best to consider `@%` as a four byte number (e.g. `@=801020903`) 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 9. 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 3 characters or spaces always follow the final E. Range of B2 in E format is 1-9.



In F format B2 specifies the number of digits to follow the decimal point. Range 0-9.

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 2 decimal places and 10 character field width.

The four bytes of @% are built up thus

@% = &	00 00 00 00
B4 – zero	00
B3 – fixed format	02
B2 – 2 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
0.005	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 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** or **OPENOUT** statements. See the section on files on page 188 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 two's 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 two's 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>}
```

## Associated keywords

```
OPENIN, OPENOUT, OPENUP, EXT#, PTR#,  
INPUT#, BGET#, BPUT#, CLOSE#
```

# 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** on page 230 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

```
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 PROCanoi(F,1,2,3)
40 END
50 DEF PROCanoi(A,B,C,D) IF A=0
ENDPROC
60 PROCanoi(A-1,B,D,C)
70 PRINT "Move disk " ;A; " from
pile " ;B; " to pile " ;C
80 PROCanoi(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 5 bytes, each real number occupies 6 bytes and each string takes up the number of letters in the string plus 2. See the entry **PRINT#** on page 328 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 open on the selected channel before this function is used. Files are opened with the **OPENIN** and **OPENOUT** statements. See page 188 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,  
OPENOUT, OPENUP, EXT#, EOF#
```

# **RAD** radian

## **Purpose**

To convert an angle measured in degrees to radians. 1 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** on page 120 for more information on arrays.

See the keyword **DATA** on page 227 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 programe

```
200 INPUT "How many pounds can you
afford" ,AFFORD
210 PRINT "You can afford the
following cars"
220 FOR X=1 TO 10
230 READ NAME$
240 READ PRICE
250 IF PRICE < AFFORD THEN PRINT
NAME$
260 NEXT
270 END
500 REM British Leyland Cars
510 DATA METRO HLE,3695
520 DATA etc etc
```

# REM remark

## Purpose

To enable the program writer to put remarks and comments into the program to help him 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 **REMs**, but as your programs grow in complexity you will find it quite essential to have them liberally sprinkled over your program. If you come back to a program six months after you wrote it and find no **REMs** you will have a real job trying to remember how it worked and why you used that variable name etc. etc. Use lots of **REMs** – it will save you hours of time in the long run.

## Examples

```
10 REM this revision dated 25-3-82
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 number 300 and 310 – difficult!

The **RENUMBER** command will go through your program and renumber it automatically. It deals successfully with 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....**

If you renumber a program containing an **ON GOTO** statement which contains a calculated line number, e.g.

**ON X GOTO 120,240,2\*R,1000,2000**

then **RENUMBER** will deal successfully with references before the calculated line number. However it will not deal with the calculated line number or other line numbers in the same statement – ie **2\*R, 1000** 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. etc.

The command **RENUMBER 200,4** would renumber starting with line 200 and then using 204, 208 etc. 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 shown on page 128. 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 at the end of a subroutine to make the computer return to the place in the program which originally ‘called’ the subroutine. See **GOSUB** on page 265 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\$ right string

## 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 eleven 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, or make, 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 brackets 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  $\mathcal{A}\%$  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 filename. The filename can have up to 10 letters and numbers in a cassette system, 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 sign

## 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,-15,53,20**

will play a note on sound channel 1, with a loudness of -15 (maximum volume). A pitch value of 53 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 (the channel number) 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 4 user-defined envelopes which are selected by setting A to be 1,2,3 or 4. Envelopes are explained on pages 182 and 244.

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 89. The table on

page 181 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 2 seconds you could enter

```
100 SOUND 1,-15,53,40
110 SOUND 2,-15,69,40
120 SOUND 3,-15,81,40
```

Whereas to play a number of notes in succession you would enter

```
100 SOUND 1,-15,97,10
110 SOUND 1,-15,105,10
120 SOUND 1,-15,89,10
130 SOUND 1,-15,41,10
140 SOUND 1,-15,69,20
```

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 2 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 page 351 later in this section.

As was mentioned earlier in this section, 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 rather rasping, harsh note. With P set to 4 the noise is not unlike that produced by a radio when it is not tuned to a station – sort of “shssh” effect. P=6 sounds like the interference found on bad telephone call.

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 one 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, whereas

```
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 need not understand exactly why the following works to use it!

For C one can write a four figure hexadecimal number to achieve certain effects – for example:

```
SOUND &1213,-15,53,40
```

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 one to synchronize notes so that one 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 amplitude and pitch defined by the rest of the **SOUND** statement is ignored.

**The second number (S)** is used to synchronize 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 (i.e. a total of 3). If a note was sent to channel one 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:

```
100 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 5 seconds (duration 100). When that 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 squarewave with programmable frequency. Channel 0 can produce noise (unpitched sound of psuedo-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 five 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 4 hexadecimal digits. Thus

**SOUND &H S F C , A , P , D**

	Range	Function
H	0 or 1	Continuation
S	0 to 3	Synchronization
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 a 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 2 more requests with 2 as the value of 'S' before playing the complete chord. Single requests use 0 for 'S' so 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 and cassette output buffers are 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 '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 space

## 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 brackets 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 5 units.

The step may be positive or negative.

**STEP** is optional, if omitted a step size of +1 is assumed – see **FOR** on page 260.

## 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 XXXX**

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 G9 with B1=0. See page 70.

The opposite function of converting a string into a number is performed by the function **VAL**.

## Examples

```
20 A$=STR$(X)
```

```
5060 num$=STR$(size)
```

## 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**



# 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 lots of 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 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), "Doctors name",
DOC$
```

**Description**

**TAB** with a single argument prints spaces (and a newline if necessary) to reach the specified column.

**TAB** with two arguments moves the cursor directly to the specified co-ordinates.

**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 the angle given.

The angle must be given in radians but may be converted to radians from degrees using the function **RAD**. 1 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. You can't use it to check the Greenwich Time Signal! 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 page 91 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 600,1000  
60 DRAW X,0  
70 NEXT X  
80 NEXT C
```

# TOP

## Purpose

The function **TOP** returns the address of the first free memory location after 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

## 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 whereas **CALL** does not. Therefore you must either assign the result to a variable

```
X=USR(&3000)
```

or print the result

```
PRINT USR(&3000)
```

## Examples

```
1400 R=USR(&3000)
```

```
670 result=USR(plot5)
```

## Description

A function allowing machine code to directly return 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$`

# 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 page 378 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 co-ordinates of the text area. The range of X is 0-39 and of Y is 0-31 in **MODE 4**. Thus

```
VDU 28,0,5,39,0
```

would define a 6 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 co-ordinates 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 co-ordinates is sent as a double byte pair. Note that the graphics origin is bottom left whereas 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 arguments are separated by commas then single bytes are sent. If any 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 newlines.

## 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 finds most common use 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.

Those writing BASIC programs will need to refer to this summary of the `VDU` drivers if they wish to use some of the more advanced facilities such as definition of graphic and text windows. Those writing other high level languages or machine code programs will also need to refer to this section.

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

## VDU code summary

Decimal	Hex	CTRL	ASCII abbrev.	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 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 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	5	define logical colour
20	14	T	DC4	0	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	re-program display character
24	18	X	CAN	8	define graphics window
25	19	Y	EM	5	<b>PL0T K, x, y</b>
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

function to be performed. The table opposite summarises all the codes and gives the number of bytes which follow the ASCII control code.

### 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 **VDU 2**. Many printers use special control characters to change, for example, the size of the printed output. For example the Epson MX-80 requires a code 14 to place it into double width print mode. This could be effected with the statement

```
VDU 1,14
```

or by pressing **CTRL A** and then **CTRL N**. This code also enables the "printer ignore" character selected by \*FX 6 to be sent to the printer.

2 This code "turns the printer on" by which is meant that all output to the screen will also be sent to the printer. In a program the statement **VDU 2** 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 **VDU 3** 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 **VDU 4**, 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

```
MOVE 700,450
```

Once the statement **VDU 5** 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 page 173 for more information.

**6** **VDU 6** is a complementary code to **VDU 21**. **VDU 21** stops any further characters being printed on the screen and **VDU 6** re-enables screen output. A typical use for this facility would be to prevent a pass-word appearing on the screen as it is being typed in.

**7** This code, which can be entered in a program as **VDU 7** 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 (**VDU 8** 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 **VDU 127**.

**9** This code (**VDU 9** or **CTRL I** or **TAB**) moves the cursor forward one character position.

**10** The statement (**VDU 10** 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 (**VDU 11** 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 **VDU 12**, 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 **VDU 13** 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 the **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 (**VDU 16** 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. **VDU 16** does not move the graphics cursor – it just clears the graphics area of the screen.

**17** **VDU 17** is used to change the text foreground and background colours. In BASIC the statement **COLOR** is used for an identical purpose. **VDU 17** is followed by one number which determines the new colour. See the BASIC keyword **COLOR** on page 222 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 used for text (and graphics) in the immediate future. However the actual colour can be re-defined with **VDU 19**. 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 3 zeros after the actual colour in the **VDU 19** 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.

*Note:* In the television series 'The Computer Programme', an attractive way of listing programs was produced by using **MODE 6** and **VDU 19,0,4,0,0,0**.



**20** This code **VDU 20** or **CTRL T** sets default text and graphic foreground logical colours 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

**Sixteen colour modes**

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

**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 **VDU 21** 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 **VDU 6**.

**22** This VDU code is used to change MODE. It is followed by one number which is the new mode. Thus **VDU 22,7** is exactly equivalent to **MODE 7** (except that it does not change **HIMEM**, see pages 270 and 301).

**23** This code is used to re-program 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 re-define *any* character that is displayed, but extra memory must be set aside if this is done. See page 427.

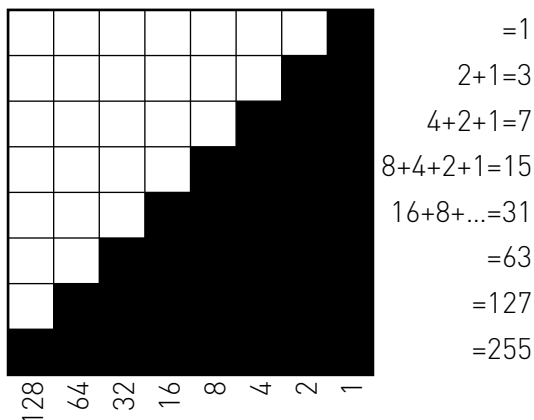
ASCII codes 0 to 31 are interpreted as VDU control codes – and this chapter is explaining the exact function of each. Thus the full ASCII set consists of all the VDU control codes, all the normal printable characters and a user defined set of characters.

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  
re-defined

VDU 23, 240, 1, 3, 7, 15, 31, 63, 127, 255

redefine character                      8 numbers giving the contents of each  
row of dots that makes up the desired  
character



Note that you cannot define your own characters in MODE 7.  
See page 170 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
```

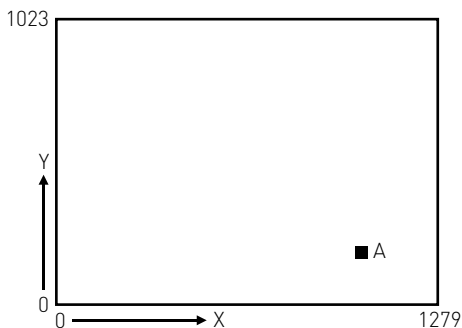
In the unlikely event of the user wishing to define more than the 32 characters mentioned above (ASCII 224 to 255) it will be necessary to allocate more RAM for the purpose. This is described on page 427

A second use of **VDU23** is to permit the advanced programmer to alter the contents of the 6845 CRTC circuit. (See page 77 for cursor control.) 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
```

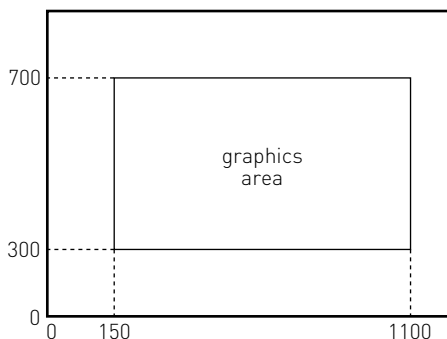
The user is cautioned not to do this unless he understands how to program the 6845. Note however that when writing to register 7 (V Sync. Posn.) or register 8 (Interlace) of the 6845, any offset that has been set up with the **\*TV** statement (page 435) will be used to adjust the value sent to R7.

**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 co-ordinates.



Thus the co-ordinates of A would be approximately 1000,200.

When defining a graphics window four co-ordinates 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 graphic co-ordinates 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. **VDU 25** is followed by 5 bytes. The first gives the value of K referred to on page 319 of the explanation of **PLOT** in the BASIC keywords chapter. The next two bytes give the X co-ordinate and the last two bytes give the Y co-ordinate. Refer to the entry for **VDU 24** for an explanation of the semi-colon syntax used. Thus

**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**  
                   X                  Y

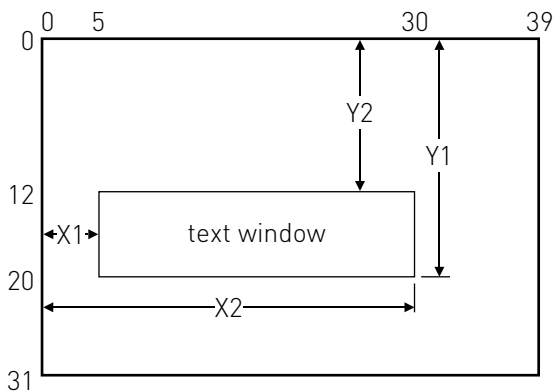
**26** The code **VDU 26 (CTRL Z)** returns both the graphics and text windows to their initial values where they occupy the whole screen. This code re-positions the text cursor at the top left of the screen, the graphics cursor at the bottom left and sets the graphics origin to the bottom left of the 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 (**VDU 28**) 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**

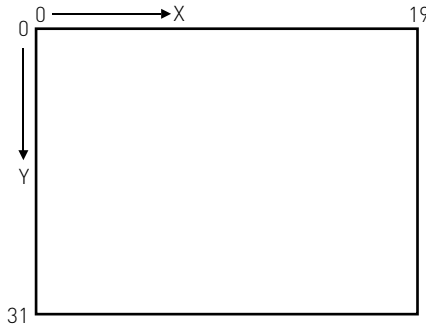
where **leftX** sets the left hand edge of the window  
**bottomY** sets the bottom edge  
**rightX** sets the right hand edge  
**topY** sets the top edge



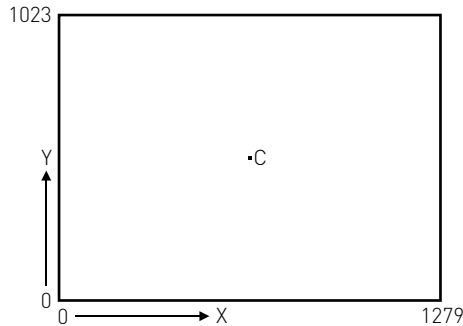
For the example shown the statement would be

**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 **MODE 1** and **MODE 4**. In **MODES 2** and **5** the maximum values would be 19 for X and 31 for Y since these modes have only 20 characters per line.



**29** This code is used to move the graphics origin. The statement **VDU 29** is followed by two numbers giving the X and Y co-ordinates of the new origin. The graphics screen is addressed



Thus to move the origin to the centre of the screen the statement

**VDU 29,640;512;**

should be executed. Note that the X and Y values should be followed by semi-colons. See the entry for **VDU 24** if you require an explanation of the trailing semi-colons. Note also that the graphics cursor is not affected by **VDU 29**.

**30** This code (**VDU 30** or **CTRL ^**) moves the text cursor to the top left of the text area.

**31** The code **VDU 31** enables the text cursor to be moved to any character position on the screen. The statement **VDU 31** is followed by two numbers which give the X and Y co-ordinates of the desired position.

Thus to move the text cursor to the centre of the screen in MODE 7 one would execute the statement

**VDU 31,20,12**

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 on pages 486 to 492.

**127** This code moves the text cursor back one character and deletes the character at that position. **VDU 127** has exactly the same effect as the **DELETE** key.

**128-223** These characters are normally undefined and will produce random shapes (see pages 384 and 427).

**224-255** These characters may be defined by the user using the statement **VDU 23**. 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 0**  
♥ **VDU 23,226,54,127,127,127,62,28,8,0**  
♠ **VDU 23,227,8,28,62,127,127,127,28,62**

*Note: on machines with operating systems after version 1.0, 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 page 439). To discover which version of the operating system you have, type*

**\*FX 0 RETURN**

# 35 Cassette files

This chapter summarises the facilities available for file handling using a cassette recorder. Refer to section 5 page 34 for an introduction to loading and saving BASIC programs.

## **A. 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 light 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.

## **B. 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.

## **C. Playback volume and tone**

It is important that the playback volume is set correctly. A detailed description of how to do this is given on page 12. The tone control should normally be set to “Maximum” or “High”.



## D. 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. It is strongly suggested that you space the programs out along the tape. Use the tape counter to start recording at positions 000, 100, 200, 300 and so on. But beware of recording on the blank leader tape – always wind it on a little first. In this way you will not only be able to find programs easily but you will leave good clear gaps between average sized programs.

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.

## E. 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 under P of this section.

3 A 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**

## F. 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 to 7 as for **SAVE** shown above.

**SSSS** represents the start address of the data, in hexadecimal.

**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 **SSSS**.

Two examples may make the syntax clearer:

**\*SAVE patch 6000 6200**

**\*SAVE match 4C00+0CE9 2A10 2000**

### **G. 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. See under Q of this section for information on dealing with the problem.

### **H. 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 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**

### **I. Loading and running a BASIC program**

The statement **CHAIN** allows a BASIC program to **LOAD** and **RUN** another BASIC program. It is particularly useful when there is a sequence of related programs, for example on the **WELCOME** cassette.

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.

### **J. 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**

### **K. 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**

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 page 402 for how to use it to merge programs.

## **L. 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 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)**

## **M. 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.

Note that **E O F #** is not implemented on release 0.10.

### **N. Storing data on tape**

The data in the last example was read into an array in the computer. It could be then edited and the corrected version could be re-recorded 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.

### **O. 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
```

## P. File-names

File-names on cassettes can be up to 10 characters long and can include any character except space.

## Q. 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  
\*SP00L

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

## R. 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 (e.g. **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.

**\*OPT 1, 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=2** gives detailed information including load and execution addresses.

**\*OPT 2, 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 re-try by re-winding the cassette.

**X=2** will make the computer abort the operation by generating an error.

**\*OPT 3, X** is used to set the interblock 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, 1** (short messages)

**\*OPT 2, 1** (prompt for re-try)

**\*OPT 3, 6** (0.6 second inter block gap)

For **BGET#**, **INPUT#**, **BPUT#**, **PRINT#**

**\*OPT 1, 0** (No messages)

**\*OPT 2, 2** (Abort on error)

**\*OPT 3, 25** (2.5 second inter block gap)



## S. Cassette tape format

The format of each block of data stored on cassette is given here for those who wish to produce tapes on other machines that may be read into the BBC computer.

5 seconds of 2400Hz tone  
1 synchronization byte (&2A)  
Filename (1 to 10 characters)  
1 end of filename marker byte (&00)  
Load address of file, 4 bytes, low byte first  
Execution address of file, 4 bytes, low byte first  
Block number – 2 bytes, low byte first  
Data block length – 2 bytes, low byte first  
Block flag – 1 byte  
Spare – 4 bytes currently &00  
CRC on header – 2 bytes  
Data – 0 to 256 bytes  
CRC on data – 2 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 filename 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 standard machines (model A and model B) are 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.

<b>*TAPE</b>	Selects the cassette filing system running at the default speed of 120 characters per second (1200 baud)
<b>*TAPE3</b>	Selects the cassette filing system running at 30 characters per second (300 baud)
<b>*TAPE12</b>	Selects the cassette file system at 1200 baud
<b>*DISC</b> <b>*DISK</b>	These two statements select the disc file system so that all future file operations (e.g. <b>LOAD</b> and <b>SAVE</b> ), work to the floppy disc units

- \*NET**                      Selects the network file system for all future file operations
  
- \*ROM**                     selects the ROM cartridge file system

# 37 How to merge two BASIC programs

There are a number of ways of merging two BASIC programs together. Two are given below. In either case the line numbers should not clash unintentionally.

In order to correctly merge two programs together it is necessary to save one of them (the shorter one preferably), as an ASCII file rather than in the usual compressed format. This ASCII version of the program is then merged into the longer program using the command **\*EXEC**. Suppose we wish to merge the program **SHORT** into the program **LONG**. First, load in the first program

```
LOAD "SHORT"
```

and then create an ASCII version by typing in the next three lines

```
*SP00L "SHORT2"  
LIST  
*SP00L
```

This produces an ASCII version of the program. The ASCII version is called **SHORT2**.

Now load in the big program by typing

```
LOAD "LONG"
```

This loads the long program

and lastly merge in the small program

```
*EXEC "SHORT2"
```

This merges the short program

The command

```
*SP00L "SHORT2"
```

informs the computer that anything that it outputs to the screen is also to be sent to a cassette (or disc, etc.) file called **SHORT2**. When the computer lists the file it therefore creates an ASCII listing on cassette. The command **\*SPOOL** without a filename 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 normal to renumber the two programs **SHORT** and **LONG** so that the line-numbers don't clash.

A rather quicker (and "dirtier") 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 predictable but 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 2 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 dialogue 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 Model B BBC computer can be used with the vast majority of printers available today. It is not possible, though, to drive old 'Teletypes' which operate at 10 characters per second, or printers requiring a 20mA current loop connection. To use a printer with your BBC computer you will need to do three things.

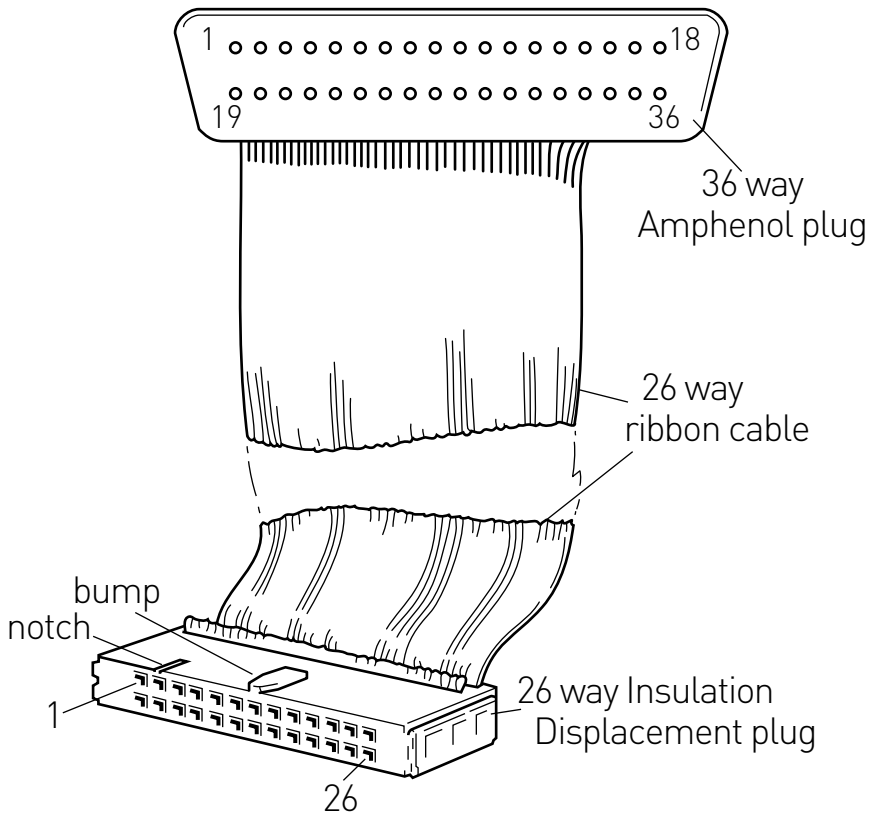
- 1 Connect the printer to the computer.
- 2 Tell the computer whether your printer is plugged into the parallel or serial printer port.
- 3 Tell the computer to copy everything sent to the screen to the printer.

## **1 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 (RS part number 467-295), 1 metre of 26-way ribbon cable (RS part number 358-062), and 1 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.

A suitable, ready made lead is available as BBC Microcomputer system part number ANG 04.



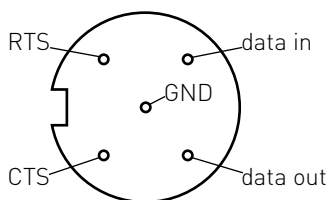
### A parallel printer cable

Both plugs viewed looking at the pins on the underside of the plug body.

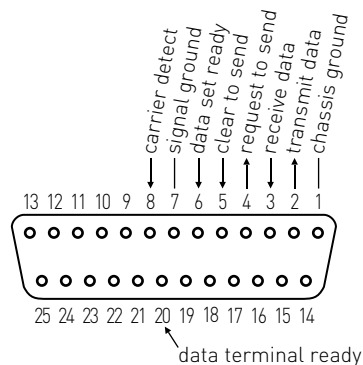
## Parallel printer connections

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

The “serial printer port” is often referred to as an “RS232-C” or “V24” port. The standard connector plug is a “Canon 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 computer emerges via a 5-pin DIN plug and it will normally be possible to drive a serial printer with just 3 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



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.

Those who require a more complete RS232 connection should try the following:

- 1 Data in to pin 2 (Transmit data)
- 2 Data out to pin 3 (Receive data)
- 3 CTS to pin 4 (Request to send)
- 4 RTS to pin 5 (Clear to send)
- 5 GND to pin 7 (Signal ground)

## **2 Tell the computer whether you are using a parallel or serial printer**

\*FX 5 is used to inform the computer whether it is to send printer output to the parallel or serial port. Type in either:

\*FX 5,1 if you are using a parallel printer

\*FX 5,2 if you are using a serial 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)

## **3 Tell 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 at the mains but rather 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 **VDU 2** will turn the printer on and **VDU 3** 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 **\*FX 3**. See page 422.

### **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 LINEFEED), 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 LINEFEED which has an ASCII code of 10. **\*FX 6** is used to set the printer ignore character, so **\*FX 6, 10** will set it to ignore "line feeds". **\*FX 6** should be set after **\*FX 5** has been used to select the printer type. The default is to ignore line feeds.

Thus 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 1 start bit, 8 data bits and 1 stop bit.

Receive baud rates may be the same or differ from transmit baud rates. Receive baud rates are set by **\*FX 7**.

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 **\*FX 5, 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 6 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 those using the computer's assembler or those wishing to create their own data structures. There are three indirection operators:

Name	Purpose	No. of bytes affected
? query	byte indirection operator	1
! pling	word indirection operation	4
\$ dollar	string indirection operator	1 to 256

For the sake of illustration let us play with 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 (pling) acts on four successive bytes. Thus

```
!M=&12345678
```

would set location

```
&2000=&78
```

```
&2001=&56
```

```
&2002=&34
```

```
&2003=&12
```

and `PRINT ~!M` (print in hex, pling M) would give

```
12345678
```

Four bytes are used to store integers so the pling 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, pling 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 pling can be used as binary operators providing 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 4 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="ABCDEFGG"  
>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 10 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 psuedo-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.

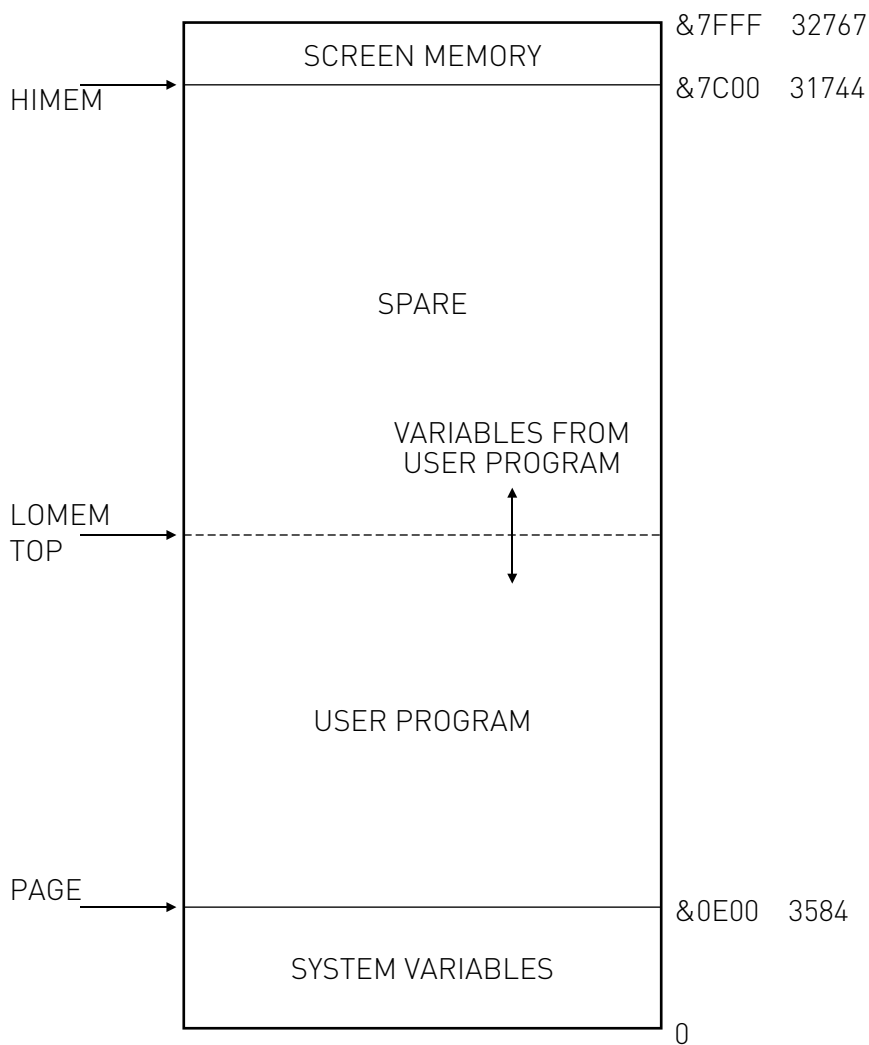
In a Model B computer Random Access Memory extends from location 0 to location 32767 (in hexadecimal that is &7FFF). RAM is normally allocated in **MODE 7** as shown opposite.

As the user enters more program so 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 re-setting 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**.

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.





# 41 Operating system statements

The computer 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 line after the asterisk is passed to the operating system directly. The operating system (O.S.) commands include,

- \*LOAD**            Loads a section of memory (not a BASIC program) (see page 393)
- \*SAVE**            Saves a section of memory (see page 392)
- \*RUN**             Loads and executes a machine code program (see page 394)
- \*CAT**             Displays a catalogue of files on the cassette/disc/net (see page 391)
- \*KEY**             Programs one of the user-defined keys (see page 141 and 439)
- \*OPT**             Determines how the computer reacts to errors during loading of cassettes and the amount of detail given during cassette operations (see page 434)
- \*FX**              Enables the user to control a large number of the computer effects such as flash rate (see page 418)
- \*TAPE**            Selects the cassette filing system running at 120 characters per second (1200 baud)
- \*TAPE3**          Selects the cassette filing system running at 30 characters per second (300 baud)
- \*DISC**            These two statements select the disc file system
- \*DISK**            so that all future file operations (e.g. **LOAD** and **SAVE**) work to the floppy disc units

- \*NET**               Selects the network file system for all future file operations
- \*SP00L**           Copies all screen output to a named file (see page 402)
- \*EXEC**             Uses a named file to provide input as if it had been typed in at the keyboard (see pages 394 and 402)
- \*MOTOR**           Can be used to turn the cassette motor relay on and 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 page 435)
- \*TELESOFT** (provisional)
- \*TERMINAL** (provisional)
- \*ROM**              Selects the ROM cartridge file system

# 42 FX Calls and OSBYTE calls

## \*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	Flush 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 synchronization
20	14	Explode soft character RAM allocation
21	15	Flush selected buffer
* 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

Decimal	Hex	Function
* 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
136	88	Reserved
* 137	89	Turn cassette motor ON/OFF
138	8A	Insert character into keyboard buffer
* 139	8B	Set file options
* 140	8C	Select cassette file system and set speed
141	8D	Reserved
142	8E	Reserved
143	8F	Reserved
* 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
218	DA	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

This sheet gives OSBYTE calls for issue 1.0.

Calls marked \* are identical in issue 0.10. Those calls with no mark are new to the issue 1.0.

The statement **\*FX** is used to control a large number of the computer's "special effects", for example the rate at which "flashing colours" flash or to select whether an RS423 or a parallel printer is to be used. For example, to select a parallel printer and to copy all output to the printer one would use the statements

```

10 *FX 5,1 selects parallel printer
20 VDU 2   copies all output to printer

```

Please note that not all the calls listed are available on operating systems before "Issue 1.0".

The FX calls are also easily available to those writing in assembly language. In this case the same effect of selecting a parallel printer and turning it on would be achieved by

```

10 DIM DEMO 20
20 OSBYTE=&FFF4
30 OSWRCH=&FFEE
40 P%=DEMO
50 [
60 LDA #5
70 LDX #1
80 JSR OSBYTE
90 LDA #2
100 JSR OSWRCH
110 RTS
120 ]
300 CALL DEMO

```

```

>RUN
0EA4
0EA4 A9 05      LDA #5
0EA6 A2 01      LDX #1
0EA8 20 F4 FF   JSR OSBYTE
0EAB A9 02      LDA #2
0EAD 20 EE FF   JSR OSWRCH
0EB0 60         RTS

```

Because this chapter is relevant to both those writing BASIC programs and those writing in assembly language, a certain amount of material has to be included that is only relevant to those writing in assembler. The next section is a case in point; it is only relevant to those programming in assembly language.

All FX calls are available in machine code. In general an FX call in BASIC takes the form

```
*FX 137,0,1
```

In assembly language the general form is

```
LDA #137
```

```
L D X  #0  
L D Y  #1  
J S R  & F F F 4
```

The first parameter (137) determines which FX call is to be used and is placed in the accumulator. The second and third parameters are optional but if used are placed on the X and Y registers.

If omitted X and Y will be set to zero. Thus **\*FX a** would set X and Y to zero and **\*FX a, X** would set Y=0.

Before using an OSBYTE call (**JSR & F F F 4**) the 6502 **must** be in binary mode. In general, after an OSBYTE call the registers and flags will be undefined. Some OSBYTE calls return a value in X. These are noted further on. OSBYTE calls with A in the range &80 to &FF in general need to set both X and Y whereas other OSBYTE calls only take one argument – in X.

As you may gather, the **\*FX** statement merely takes the parameters given, places them in A, X and Y and executes a call to &FFF4.

Some FX statements return values in the X (and sometimes Y), register. They can only be used by assembly language programmers or by using the **USR** function in BASIC. For that reason they are explained as OSBYTE calls rather than FX calls.

See section 43 (Assembly Language) for more information.

Each **FX** and OSBYTE call is now explained in detail.

**\*FX0** prints a message giving the version number of the operating system.

**\*FX1** is reserved for user supplied routines in application programs.

**\*FX2** selects the input device from which characters will be fetched.

**\*FX2, 0** get 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.

This call is not available in operating systems prior to issue 1.0.





\*FX3 is used to select whether output appears (or not), on three different output streams. In the table below *screen* refers to the normal TV or monitor, *RS423* refers to the RS423 socket and *Printer* refers to the selected printer output which may be either parallel or RS423.

The number following \*FX3, will produce output on the following devices





	<i>Printer</i> (bit 2)	<i>Screen</i> (bit 1)	<i>RS423</i> (bit 0)
*FX3,0	✓	✓	x
*FX3,1	✓	✓	✓
*FX3,2	✓	x	x
*FX3,3	✓	x	✓
*FX3,4	x	✓	x
*FX3,5	x	✓	✓
*FX3,6	x	x	x
*FX3,7	x	x	✓

Thus, to send output to the *RS423* but not to the *screen* or *printer* one would use \*FX3,7





At machine code level bit 0 controls the RS423 stream, bit 1 the VDU screen and bit 2 the printer. Note that bits 1 and 2 are negative logic true so that \*FX3,0 will select the printer and screen outputs. This is the default setting.





\*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.

<b>COPY</b>	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 number are

<b>COPY</b>	11
	12
	13
	14
	15

This feature is only available from issue 1.0.

\*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 page 408).

Note that neither of these will actually cause output to appear on the printer. VDU 2 (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 “linefeed” 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 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.

Note that the default is that line feed characters are filtered out.

\*FX6,0 filters no characters.

\*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 a 1 start bit, 8 data bits, 1 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 page 222 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 fulltime. 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 centi-seconds.

**\*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 enable and disable certain "events". These FX calls will only be used by those writing in assembly language and more information on "Events" is given on page 464. The summary which follows should be read in conjunction with that section.

**\*FX13,0** disable output buffer empty event.

**\*FX13,1** disable input buffer full event.

**\*FX13,2** disable character entering input buffer event.

**\*FX13,3** disable ADC conversion complete event.

**\*FX13,4** disable start of vertical sync. event.

**\*FX13,5** disable interval timer crossing zero event.

**\*FX13,6** disable **ESCAPE** pressed event.

These calls are only available from issue 1.0.

- \*FX14,0 enable output buffer empty event.
- \*FX14,1 enable input buffer full event.
- \*FX14,2 enable character entering input buffer event.
- \*FX14,3 enable ADC conversion complete event.
- \*FX14,4 enable start of vertical sync. event.
- \*FX14,5 enable interval timer crossing zero event.
- \*FX14,6 enable **ESCAPE** pressed event.

These calls are only available from issue 1.0.

The initial condition is that all are disabled.

\*FX15 enables various internal buffers to be flushed (or emptied).

\*FX15,0 flushes all internal buffers.

\*FX15,1 flushes the currently selected input buffer.

See also \*FX21

As an OSBYTE call this typically executes in 370 micro-seconds in release 1.0.

\*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 only enable 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)**.

This call is only implemented from issue 1.0.

**\*FX18** Resets the user defined function keys so that they no longer contain character strings.

This call is only implemented from issue 1.0.

**\*FX19** Causes the machine to wait until the start of the next frame of the display for animation. This call is only implemented from issue 1.0.

**\*FX20** When the machine starts up, space is allocated at &C00 for the re-definition of 32 displayed characters using the **VDU 23** statement.

In operating system 0.10 ASCII characters 224 to 255 can be redefined into this space. In operating system 1.0 a new concept is introduced namely that of “exploding” the memory allocation for user defined characters. What follows refers only to version 1.0 but care has been taken to ensure upward compatibility with operating system 0.10.

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 it will then appear at four ASCII positions due to this mapping. Thus ASCII codes &80, &A0, &C0 and &E0 will all show the same character.

Re-defining ASCII code &E0 (224) will affect all four characters. Codes between &20 and &7F can be re-defined but will map to memory at &C00, thus overwriting what is already there.

After a **\*FX20,1** command the character definitions are said to be “exploded” and all printing characters (codes &20 to &FF) can be re-defined. However, until any character in a block is re-defined, the block will map on to either the pre-set ROM (&20 to &7F) or on to memory at &C00 (&80 to &FF). The memory allocated to each block of ASCII codes in the “exploded” state is

<b>ASCII code</b>	<b>Memory allocation</b>
&20 to &3F	OSHWIM + &300 to OSHWM + &3FF
&40 to &5F	OSHWIM + &400 to OSHWM + &4FF
&60 to &7F	OSHWIM + &500 to OSHWM + &5FF
&80 to &9F	&C00 to &CFF
&A0 to &BF	OSHWIM to OSHWM + &FF
&C0 to &DF	OSHWIM + &100 to OSHWM + &1FF
&E0 to &FF	OSHWIM + &200 to OSHWM + &2FF

Thus if the user re-defines characters in the range &20 to &3F he must leave memory up to OSHWM + &3FF clear of his program. This is done by first finding the value of OSHWM (for the particular configuration in use) by using OSBYTE call 131 and when setting **PAGE** to a value &400 above this value. Of course, if the user only wishes to re-define ASCII codes in the range &80 to &9F (128 to 159) then he need not alter **PAGE**.

Codes in the range &80 to &9F are of particular importance since the user can generate them directly from the keyboard by holding down **SHIFT** at the same time as pressing one of the user definable function keys (issue 1.0 only).

See page 431 for more information on Operating System High Water Mark.

\* **FX20** is only available from issue 1.0.

\* **FX21** This call allows any internal buffer to be emptied (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** and OSBYTE &80

This call is only implemented from issue 1.0.

\* **FX124** Can be used to reset the **ESCAPE** pressed flag at location &FF. It must not be set or reset directly.

\* **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.

### **OSBYTE calls**

**Most of the calls that follow can only be used in machine code programs.** They are therefore described in terms of an OSBYTE call – that is, a JSR to location &FFF4. The call number is passed in the A accumulator and the X and Y registers are used to pass other values. Values are returned in X and/or Y. It is however possible to use these calls from BASIC by using the USR function, provided that the result is deposited in an integer variable.

#### **OSBYTE 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-of-file has been reached.

#### **OSBYTE 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 8 most significant bits and X the 8 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 2’s complement notation) then the call will provide information about various input buffers. On entry Y must contain &FF.

<b>X on entry</b>	<b>Buffer checked</b>
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 with A=&81 (129) Read key with 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 centi-seconds and Y contains the most significant byte and X the least significant byte. The maximum delay is &7FFF centi-seconds which is about five-and-a-half minutes.

On exit the 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 micro-seconds in release 1.0.



### **OSBYTE with A=&82 (130) Read machine higher order address**

The BBC computer uses a 6502 processor 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 computer. This call generates the high order address for each machine. The high order address is returned in X and Y with Y containing the most significant byte and X the least significant byte.

### **OSBYTE with A=&83 (131) Read top of O.S. 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. In a cassette system this is normally &E00. The value is returned in X and Y with Y containing the most significant byte and X containing the least significant byte.

```
10 A%=131
30 PRINT ~USR(&FFF4)
```

```
>RUN
B10E0083
```

&0E00 is the value returned, &83 is the OSBYTE code.

### **OSBYTE 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. Its value depends, among other things, on the selected display mode and on whether the machine has 16K or 32K of RAM. As usual, Y contains the most significant byte and X the least significant byte of the result.

### **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 modes, it merely investigates the possible consequences of changing mode.

### **OSBYTE call with A=&86 (134) Read text cursor position**

This call returns in X and Y the X and Y co-ordinates of the text cursor. A similar function is performed in BASIC by **POS** and **VPOS**. As an OSBYTE call this takes typically 100 micro-seconds in release 1.0.

### **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 is only implemented in version 1.0 and takes typically 120 micro-seconds.

## **OSBYTE call with A=&88 (136) Reserved**

### **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 in addition in version 1.0 it 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 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 keyboard buffer**

This call enables characters to be inserted into the keyboard buffer. On entry X=0 and Y contains the character to be inserted. Thus to place the letter "R" (ASCII code 82) into the keyboard input buffer one would use

**\*FX138,0,82**

In machine code the X register must contain the buffer number and the Y register the character. e.g.

```
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
0E82
0E82 A9 8A      LDA #138
0E84 A2 00      LDX #0
0E86 A0 52      LDY #82
0E88 20 F4 FF   JSR OSBYTE
0E88 60 RTS

```

This is an extremely powerful facility! This call is only implemented from release 1.0.

### **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 page 398 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 &FFFF4

```

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=&8D (141) Reserved**

**OSBYTE call with A=&8E (142) Reserved**

**OSBYTE call with A=&8F (143) Reserved**

**OSBYTE call with A=&90 (144) TV**

This call is functionally equivalent to **\*TV**. The contents of the X register is used to control the vertical position of the screen display. For example, setting X=2 would move the display 2 text lines up the screen. Setting X=253 would move the display 3 lines down the screen. The contents of Y should be either 0 or 1 on entry. Y=0 permits an interlaced display Y=1 causes 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 cannot be turned off in **MODE 7**.

**OSBYTE call with A=&91 (145) Get character from buffer**

This call 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 computer), or in a second processor. If the user insists on addressing I/O ports directly (e.g. **STA &FE60**) then he or she will have to re-write 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 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. The calls are not available before ROM issue 1.0.

An application note entitled "BBC Computer 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 on page 437.

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 is given in the section which deals with Input/Output. See page 467.

## SHEILA addresses (offset from &FE00)

Hex. offset	Integrated Circuit	Register Designation	Description	
			Write	Read
00	6845 CRTC	STATID	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	
18	LS244			Econet station no.
20	Video ULA			
21	Video ULA			
30	LS161		Paged ROM I.D.	
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"	
64	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 Reg 1 Except No "Handshake"	
80	8271 FDC	CR1/SR1 CR2/SR2 TxFIFO/ Rx FIFO Tx FIFO/ Rx FIFO	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
A0	68B54 ADLC		Control register 1	Status register 1
A1	68B54 ADLC		Control register 2/3	Status register 2
A2	68B54 ADLC		Transmit FIFO, continue	Receive FIFO
A3	68B54 ADLC		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 &FFFF4
```

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=&DA (218) Cancel VDU queue**

Many VDU codes expect a sequence of bytes. For example, **VDU19** should be followed by 5 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 read or 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.

This facility is only available from issue 1.0.

## 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 in release 1.0 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-colour Teletext codes. The default setting is

**\*FX226,128**

On entry Y must contain zero.

This facility is only available from release 1.0.

### **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 output to the screen going into the next line.

### **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.

This facility is not available before issue 1.0.

### **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.

**\*FX230,0** will permit all buffers to be flushed

**\*FX230,1** will ensure that no buffers are flushed

This facility is only available from issue 1.0.

## **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.

In a Model A the mask byte defaults to &00 and in Model B machines the default is &FF.

This facility is only available from issue 1.0.

## **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 on both Model A and Model B.

This facility is only available from issue 1.0.

# 43 Assembly language

Why use an assembly language rather than BASIC?

BASIC is a relatively easy language to learn and another of its great strengths is that when you make a mistake the computer tries to give you some helpful message explaining what you have done wrong – or what it is incapable of doing.

But there are prices to be paid for this “friendliness” and sometimes one has to communicate with the computer in its own language (machine code) or a low-level language like assembly language – despite all the problems this can bring. Assembly language consists of statements like

**LDA #3**

which means “Load the accumulator with 3” or

**JSR &6000**

which means “Jump to the subroutine at location 6000 hex”

An Assembler is a computer program which converts these assembly language “mnemonics” into the machine code numbers that the computer actually understands.

Assembly language is used where high speed is vital 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 requirements. There are many other computer languages each with their own strengths and weaknesses.

It is unfortunately not possible, in a book this size, to teach you to write in assembly language, even though the BASIC interpreter contains an assembler. However those familiar with 6502 assembly language will need a good deal of information to enable them to write effective programs. The rest of this section explains, for those familiar with 6502 assembly, the main features of the machine code access routines.

As the reader will be aware 16K of ROM is set aside for the Machine Operating System (MOS). The MOS looks after all input/output and is accessed via a set of well defined entry points, many of which are vectored via RAM.

Access is provided to all graphics and text features including MODE selection, line drawing, text and graphic windowing, redefinition of characters, 6845 CRTC registers, flash rate control.

Access is also provided to sound generation, clock set/read, parallel/serial printer I/O, baud rate selection, ADC results and event completion flags.

The machine code programmer may select a different file system (tape, disk or net) and may perform a wide range of file input/output to the selected file system.

Considerable efforts have been made to give the assembly language programmer access to entry points. Professional programmers will understand the need to use the entry points provided. These will remain static and will work in all configurations. The assembler is part of the BASIC interpreter and can be entered at any time by placing a left hand square bracket [ as a BASIC statement. The end of the assembly language section is indicated by a right hand square bracket ]. Note that in MODE 7, the Teletext mode, these symbols are displayed as left and right arrows ← and → respectively.

```
10 PRINT "THIS IS BASIC"  
20 [  
30 JSR &FFE7  
40 ]
```

(Do not try this program – it is incomplete and could prove ‘fatal’)!

In the example given the assembler has been given no indication of where to store the assembled machine code. The “program counter” is set by the variable P% and this can be set to any desired value e.g.

```
10 PRINT "THIS IS BASIC"  
15 P%=&7000  
20 [  
30 JSR &FFE7
```

```

40 ]
50 PRINT "AND THIS IS BASIC TOO"

>RUN
THIS IS BASIC
7000
7000 20 E7 FF JSR &FFE7
AND THIS IS BASIC TOO

```

However another, and much more useful, way is provided to enable the programmer to allocate a few bytes of memory to a piece of machine code. A special version of the **DIM** statement is used.

```

10 PRINT "THIS IS BASIC"
15 DIM GAP% 20
16 P%=GAP%
20 [
30 JSR &FFE7
40 ]
50 PRINT "AND THIS IS BASIC TOO"

>RUN
THIS IS BASIC
0E74
0E74 20 E7 FF JSR &FFE7
AND THIS IS BASIC TOO

```

The above program dynamically allocates a gap of 21 bytes and the address of the start of the block is given in the variable **GAP%**. This is the safe way of inserting machine code sections into BASIC programs. Of course the actual location of the routine will change if more BASIC statements are inserted in the program before the **DIM** statement.

```

10 PRINT "THIS IS BASIC"
12 REM INSERT AN EXTRA LINE
15 DIM GAP% 20
16 P%=GAP%
20 [
30 JSR &FFE7
40 ]
50 PRINT "AND THIS IS BASIC TOO"

```

```

>RUN
THIS IS BASIC
0E8F
0E8F 20 E7 FF JSR &FFE7
AND THIS IS BASIC TOO

```

Two BASIC statements provide access to machine code routines: **CALL** and **USR**. The **USR** statement provides an easy way of passing values to and from the microprocessor's registers before and after the machine code call. When the statement

```
R = USR(Z)
```

is executed the computer first transfers values from the BASIC variables **X%**, **Y%**, **A%** and **C%** into the processor's X and Y register, accumulator and carry flag respectively. Of course, only the least significant byte of **X%**, **Y%** and **A%** can be used and only the least significant bit of **C%** can be copied into the carry flag.

After the various registers have been set up, control is passed to the machine code subroutine (at address Z). This routine *must* end with an **RTS** if control is to return to BASIC. On return the A, X and Y registers and the program status register (P) are used to produce a number which is placed in the BASIC variable R.

This can be neatly illustrated by a short program which uses an **OSBYTE** (Operating System Byte) call to determine the position of the text cursor on the screen.

```

30 A%=&86
40 X%=0
50 R=USR(&FFF4)
60 PRINT ~R

```

Lines 30 and 40 ensure that the accumulator and X register will be set up correctly. Line 50 obtains the result in R and line 60 prints the result in hexadecimal. A typical result would be

```
B1120086
```

**B1** has come from the program status register (P)  
**12** from the Y register which gives the Y co-ordinate of the cursor  
**00** from the X register giving the X co-ordinate  
**86** from accumulator which was set up from A%

The **CALL** statement provides much more flexible access to machine code routines. **CALL** does not return a calculated result into a BASIC variable. However, the user can, if he or she wishes, make any BASIC variable or group of variables available to the machine code routine. If a list of BASIC variables follows the **CALL** statement then a "parameter block" will be set up starting at location &600. The sole purpose of this block is to let a machine code routine know where variables are stored in memory and what type each variable is.

The demonstration program which follows shows the use of the **CALL** statement and also of the error handling features.

```

10 REM Demonstration of CALL with a
parameter
20 PROCINIT
30 A%=&12345678
40 CALL SWAP,A%
50 PRINT~A%
60 END
70 DEF PROCINIT
80 DIM Q% 100
90 FOR C=0 TO 2 STEP 2
100 PAR=&600: REM Parameter block
110 ZP=&80: REM Usable zero page
120 P%=Q%
130 [OPT C
140 .SWAP LDA PAR
150 CMP#1:BEQ OK
160 .ERRO BRK
161 ]
162 ?P%=45
163 $(P%+1)="Swap Parameters"
164 P%=P%+16
170 [OPT C:BRK
180 .OK LDA PAR+3:CMP #4:BNE ERRO
190 LDA PAR+1:STA ZP
200 LDA PAR+2:STA ZP+1
210 LDY#3:LDX#0
220 LDA(ZP,X):PHA:LDA(ZP),Y:
STA(ZP,X):PLA:STA(ZP),Y
230 RTS
240 ] NEXT
250 ENDPROC

```



Lines 10 to 60 form the main BASIC program. This uses a machine code program to swap parts of an integer number around. Line 20 calls a procedure which assembles a piece of machine code. Line 30 sets A% to some arbitrary value and then line 40 calls the machine code subroutine to swap A% around. Line 50 shows A% after it has been “vandalised”.

The procedure itself (lines 70 to 250) assembles the assembly language mnemonics into machine code. Line 80 tells the computer to allocate 101 bytes of memory and to set Q% to the address of the memory allocated. Lines 90 and 240 form a **FOR . . . NEXT** loop which makes the computer assemble the code twice – the first pass to work out the addresses of the labels, and the second pass to put them into code. Line 120 sets the program counter at the start of each of the two passes. Lines 140 and 150 check to see that the value in location &600 is 1. Location &600 contains the “number of parameters” and our machine code routine is only meant to have one parameter. See page 214 for information about parameter passing from **CALL**. If location &600 does not contain 1 then control will pass to the routine labelled **ERR0** (line 160).

This routine shows how a user supplied routine can produce standard BASIC error messages and error numbers. This is explained further on page 464. Suffice it to say that here the code will generate BASIC error number 45 (a new one) and will issue the message **Swap Parameters** before returning control to BASIC. Line 164 increments the internal program counter enough to allow space for the messages etc.

Line 170 re-enters normal assembler operation and re-sets the **OPT**. Line 180 checks to see that the parameter is of type 4 (an integer variable – see page 66) and if not reports an error. Lines 190 to 220 perform the actual swap and line 230 ensures an orderly return to BASIC.

### **The BASIC assembler**

The assembler is entered with a statement starting with **[** and exited with **]**. In Teletext mode these characters are displayed as ‘arrow left’ and ‘arrow right’. In the assembly language section the pseudo-operation **OPT** can be used to control listing and error reporting. See the keyword **OPT** for more details, in brief:

OPT 0 gives no errors and no listing  
 OPT 1 gives no errors and a listing  
 OPT 2 reports errors but gives no listing  
 OPT 3 reports errors and gives a listing.

Normally the "first pass" should be with OPT 0 and the "second pass" with OPT 2. If a listing is required then the "second pass" should be OPT 3.

**Labels** are defined by preceding them with a full stop. No full stop is required when they are referenced.

```

      LDA #8
.LOOP BIT VIA+&D
      BNE LOOP

```

Labels are normal BASIC variable names. Thus they should start with a letter and not start with a reserved word. During the "first pass" unknown labels will generate an address of the current op-code. Thus JMP and branch instructions will jump or branch to themselves.

To enter **byte or string constants** the programmer should leave the assembler temporarily and use indirection operators. For example

```

JMP LABEL1
]
$P%="TODAY"
P%=P%+6
[
.LABEL1 LDA&

```

Remembering that the BASIC variable P% is used to store the value of the program counter, the string indirection operator (\$) can be used to insert ASCII characters directly into memory. The word TODAY will be copied to memory followed by a carriage-return (&0D), a total of 6 characters. If you wish to follow the word by a byte containing zero instead of &0D you could write

```

$P%="TODAY"
P%?5=0
P%=P%+6

```

To insert numeric values directly into memory you can use byte or word indirection operators:

```

JMP LABEL2
]
?P%=10
P%?1=20
P%?2=66
P%=P%+3
[

```

The various 6502 addressing modes are indicated in assembly language in the standard way, but using an ampersand to represent a hexadecimal number.

Addressing mode	Examples	
Immediate	LDA #4	LDA #VALUE
Absolute	JMP &8000	LDA LOCATION
Zero page	LDA &FC	STY TEMPI
Accumulator	ASL A	
Implied	RTS	
Pre-indexed indirect	LDA (5,X)	CMP (OFFSET,X)
Post-indexed indirect	LDA (&84),Y	ORA (TABLE),Y
Zero page X	LDA 10,X	INC CNT,X
Absolute X	LDA &1234,X	STA STORE,X
Relative	BNE &3210	BNE LOOP3
Indirect	JMP (&20E)	JMP (WRCHV)
Zero page indexed with Y	LDX 24,Y	LDX VAR,Y

**Comments** can be inserted into assembly language by preceding them with a backslash. On the keyboard and in modes 0 to 6 this is shown as \ whereas in Teletext mode it is displayed as one-half (½). In BASIC everything after a REM statement is ignored on the current line so it is not possible to put another statement on the same line. Thus

```
10 REM HELLO : PRINT "FRIDAY"
```

would not print the word FRIDAY. However, in assembly language a comment terminates at the end of the statement. All the following statements would be assembled

```

LDA #&10 \MASK :.LOOP BIT VIA+13
\INTERRUPT FLAG :BEQ LOOP

```

## **Machine code entry points**

The BBC computer 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 computer) 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 the 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 re-write your code when you move it to the second processor.

## **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 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 &E1DC which is the local internal address of the normal “output character in A” routine.

Parameters are passed to the routines in various ways using either the 6502 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 (i.e. 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.

On page 452 is a summary of operating system calls and indirection vectors.

## FILES

Files are treated as a sequence of 8 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**.

## 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
OSFIND	FFCE	FINDV	21C	Open or close a file
OSGBPB	FFD1	GBPBV	21A	Load or save a block of memory to a 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 Y=0 then the operating system was unable to open the file.

If A=0 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 a file which has been opened with OSFIND. The routine is entered at &FFD1 and vectors via &21A. It is not available on cassette systems, and is documented with the disc system.

### **OSBPUT**

Writes (puts) a 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      will return, in A, the type of file system in use. The value of A on exit has the following significance

- 0   no file system
- 1   1200 baud cassette file system
- 2   300 baud cassette file system
- 3   ROM pack file system
- 4   Disc file system
- 5   Econet file system
- 6   Teletext/Prestel Telesoftware file 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 01	Address of file name, which must be terminated by &0D	LSB MSB
02 03 04 05	Load address of file	LSB   MSB
06 07 08 09	Execution address of file.	LSB   MSB
0A 0B 0C 0D	Start address of data for write operations, or length of file for read operations	LSB   MSB
0E 0F 10 11	End address of data, that is byte after last byte to be written or file attributes.	LSB   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
- A=2      Write the load address (only) for the named file
- A=3      Write the execution address (only) for the named file
- A=4      Write the attributes (only) for the named file
- A=5      Read the named file's catalogue information. Place the file type in A
- A=6      Delete the named file
- 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 be 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 8 bits have the following meanings

Bit	Meaning	
0	readable by you	(DFS only uses bit 3: file locked)
1	writable by you	
2	executable by you	
3	not deletable by you	
4	readable by others	
5	writable by others	
6	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 page 464 which deals with BRK handling.

**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 page 421.

**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	0A	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 a 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 page 422 for more details. OSWRCH is entered at location &FFEE and indirections 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 4 bytes which defines a text window. See the section 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 available in release 1.0 of the operating system.

### **OSWORD summary**

<b>A=</b>	<b>Summary of function</b>
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 internal timer
4	Write internal 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 5 bytes

00	Address of buffer for input line	LSB
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 5 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 5 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 5 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 5 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 5 locations which contain the new value to which the clock is to be set. The interval timer increments and may cause an event (see page 464) when it reaches zero. Thus setting the timer to &FFFFFFFFFE would cause an event after 2 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 8 bit byte will be stored in XY+4.

**OSWORD call with A=6**      Write to I/O processor memory

As pointed out elsewhere, programs that are to work through the Tube<sup>®</sup> 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

**OSWORD call with A=7        Make a sound**

This call can be used to generate a sound. The 8 bytes pointed to by locations XY to XY+7 are treated as four 2-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 2 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 5 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 co-ordinate
XY+1	MSB of X co-ordinate
XY+2	LSB of Y co-ordinate
XY+3	MSB of Y co-ordinate

**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 8 by 8 matrix of dots. The pattern of dots for each character in MODES 0 to 6, including user-defined characters, is stored as 8 bytes (see page 384). This call enables the 8 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 a physical (or displayed) colour associated with it. The physical to logical association can be changed with **VDU 19**. This **OSWORD** call enables one to determine the physical colour currently assigned to each logical colour.

**On entry** the X and Y registers contain the address of the start of a block of 5 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 **VDU 19** assigned a physical 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	physical colour (blue)
XY+2	0	
XY+3	0	padding zeroes 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 indirections 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 &E0A in which case successive bytes would contain:

Address	Contents	
& E0A	4 D	(M)
& E0B	4 F	(O)
& E0C	5 4	(T)
& E0D	4 F	(O)
& E0E	5 2	(R)
& E0F	2 0	(space)
& E10	3 1	(1)
& E11	0 D	(return)

Of course, this particular example would have been easier as an **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 6502 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 indirecs 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 fault 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 program on page 446 shows this in practice. See also page 466 (IRQ).

Whilst 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 processor’s 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 operating system ROM is fitted NMI’s 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 indirections through location &204 (IRQ1V) to an operating system routine which handles all anticipated internal IRQ's. 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 indirections through &206 (IRQ2V). Thus the user routine for handling IRQ's should normally be indirectioned via IRQ2V but if top priority is required the user routine can be indirectioned 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 IRQ's 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 indirections 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.

This facility is only available from release 1.0.

# 44 Analogue input

Model B computers are fitted with a socket at the back marked "analogue in". Into this socket you can plug "games paddles" and "joy-sticks" as well as voltages which the computer can measure. Games paddles usually consist of a box with a knob like a record player volume control. The computer can tell the position of the games paddle and so it can be used in games and more serious programs, for example, to move things around the screen. Joy-sticks, 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 games paddles and joy-sticks can be fitted with push buttons and the computer can detect when these buttons are pressed. The BBC computer can be connected to 4 games paddles or 2 joy-sticks. 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. 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 twelve 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/Output using the 8 bit user port**

Model B of the BBC computer contains an 8 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 Adaptor 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 twenty 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<sup>®</sup>. 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 8 bits of X to the 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 mapped 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 page 436) 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 reg.
160 JSR &FFF4 \ Call OSBYTE to get
value

```

And next the 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 reg.
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 reg.
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.



# 45 Expanding the system

Above all the BBC computer is an expandable system. The Model A can be expanded to the Model B at any time. A **floppy disc** interface and one or more floppy disc drives can be added later. A floppy disc unit will enable you to load and save programs in a matter of a second or two. If you would like to keep a mailing list, or any large data-base (like lists of products and suppliers) then you will find floppy discs invaluable since they work so much faster than cassette tapes.

For many applications you will want to add a **printer**. There are many types and the Model B can work with all except very old, slow printers such as the 10 characters per second Teletypes®.

Choosing a printer can be a difficult decision. A booklet from the Council for Educational Technology, 3 Devonshire Street, London, W1N 2BA called "USPEC 32B" provides much impartial guidance. A Daisy Wheel printer will produce high quality print at a relatively high price. On the other hand, a dot-matrix or ink-jet printer is a great deal cheaper and quite adequate for many jobs. The BBC computer can connect to RS232 serial printers or Centronics® type parallel printers.

The high quality **colour monitor** will enable you to see the finest detail produced by the computer's high resolution colour circuits.

The **Teletext add-on unit** enables the BBC computer to look at the BBC and ITV CEEFAX, and ORACLE pages in the normal way. In addition, with this unit you can receive and store, on cassette or disc, computer programs that are transmitted over the air free of charge (so called *Telesoftware*).

The **Prestel add-on unit** connects the BBC computer to a British Telecom telephone line. Once connected, you can use all the facilities of Prestel and, again, load computer programs supplied via Prestel. The transmission of computer programs in this way is known as Telesoftware. The Prestel unit contains an approved modem with which you can communicate with large computer systems world wide as well as with anyone else who has a BBC computer and Prestel unit. In this way you can send programs, or sales results or whatever, to anyone else in the world, at very low cost.

The **8 bit user input/output** port on the Model B can be connected to a wide range of external devices. These include: **bit-pads** which enable you to trace or sketch pictures on to the screen. These can be used when entering engineering drawings into the system. **Robot arms** for computer controlled movement, **video disc units** enable one of a million pictures to be rapidly retrieved, and **fast analogue interfaces** enable the computer to record fleeting events.

There is space inside the computer to add the **voice synthesis unit**. This gives the computer a vocabulary of over 100 words and further words can be built up or loaded in from cassettes or disc or ROM-pack. It is very easy to use this unit.

The **ROM-pack interface** can be added to enable the computer to use programs held in convenient cartridges. This makes it very easy to play games and to get a particular applications program working.

The **Econet option** will allow you to connect over a hundred BBC computers together. A normal Econet system consists of several Model A or Model B systems with Econet interfaces connected to a printer via a printer-server computer and to a floppy disc via a file-server computer. The only additional costs are for cable, plugs and sockets and for the file-server software. This is an extremely cost-effective way of providing printer and floppy disc facilities for a group of users. Each user on the Econet can communicate with other users (unless barred by a supervisor), and there are many other facilities. For example, one user can request a copy of another user's screen.

The computer can be interfaced to larger and faster networks such as the Cambridge Ring, thus opening up possibilities for ultra fast local networks and true distributed processing.

The **1MHz bus** connector is provided to enable users and other manufacturers to connect additional specialist hardware to the BBC computer. For example, a range of Eurocards and a racking system are available. The Eurocards include laboratory interfaces, additional parallel and serial interfaces, EPROM programmers and fast analogue to digital converters. The Prestel and Teletext units also connect to this “bus”.

The **Tube<sup>®</sup>** is a connector and software interface which permits very high speed communication between the BBC computer and a second processor. The “second processor” is another computer system in its own right, but one that uses the BBC computer for all the time-consuming input/output operations.

The first “second processor” is a fast 6502 with 64K RAM. This should be able to run any program that a Model A/B can run but with increased speed and with more space available to the user. Of course only programs that adhere to the Tube<sup>®</sup>'s software discipline will run correctly.

Another second processor option is a Z80 with 64K of RAM. The software with this unit allows CP/M<sup>®</sup> programs to be run with more memory than a normal CP/M environment. In addition, the main user program is left free to do calculations, leaving the BBC computer to deal with graphics, printers, clock, floppy disc, etc.

A future second processor will be a 16 bit machine with 32 bit internal architecture. This processor, the National semiconductors 16032, can be used with up to 16 Megabytes of RAM.

As you will be aware, there is very considerable room for expansion in the future – an essential feature in view of the rapid changes in technology.

# 46 Error messages

If the computer is unable to proceed for some reason then it will report that fact to you by printing an error message on the screen. The printing of the message can be suppressed by the statement, for example

```
ON ERROR GOTO 9000
```

provided that some special routines had been written in line 9000. 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.

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**. e.g.

```
PRINT SIN(10000000)
```

## **Arguments                      31**

This error indicates that there 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.  
e.g.

```
PRINT &y
```

**Bad MODE** 25

This indicates an attempt to change **MODE** inside a procedure or function, or to select a mode for which there is insufficient memory.

**Bad program**

There are a number of occasions on which the computer checks to see where the program that it contains starts and ends in memory. The untrappable 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 only loading part of a program or by overwriting part of the 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.

**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.

**Byte** 2

An attempt was made, during an assembly language section, to load a register with a number requiring more than one byte e.g.

```
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 is 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 the tape along with other internal information – see page 399 for more 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 by zero cannot be done! e.g.

**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 less than 100 hex. Thus this is illegal:

**\$40="hello"**

**EOF                          223**

This error is generated by the cassette filing system if 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 0**

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 would not be able to deal with

```
133 GOTO 140  
200 END
```

If the user attempts to renumber this program he will get the error message

**Failed at 10**

and the renumbered program will be

```
10 GOTO 140  
20 END
```

**File?** 219

This error generated by the cassette filing system 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 is an error generated by the cassette filing system and it indicates that a header Cyclic Redundancy Check error has occurred – see page 399 for further information. 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. e.g. `LDA Z,Z`.

**Key in use 250**

An attempt has been made to define a function key while another function key is being expanded. e.g.

`*KEY 0 *KEY 1 RUN ;M ;M`

**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. e.g. `PRINT LOG(-10)`

**Missing , 5**

This error indicates that the computer expected to find a comma in the line, and didn't do so. e.g. `D$=MID$(A$)`

**Missing " 9**

The computer expected a double quote. e.g. `LOAD "FRED`

**Missing ) 27**

The computer expected a closing bracket. e.g. `TAB(10,10`

**Missing # 45**

The computer expected a hash. e.g. `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. e.g. `SQR(-10)`. May also occur with `ASN` and `ACS`.



**No GOSUB 38**

A **RETURN** statement was found when no **GOSUB** statement had been encountered.

**No FN 7**

This indicates that the computer detected the end of a function but had not called a function definition. e.g.

= **FN sue**

**No FOR 32**

A **NEXT** statement was found when no **FOR** 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 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** (e.g. **FNfred**) or **PROC** (e.g. **PROCsteve**) it expects to be able to find a corresponding function or procedure definition somewhere. This error indicates that no matching definition 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 this will fail.

```
ON X GOTO 100,200
```

since there are only two destinations.

**ON syntax 39**

The **ON . . . GOTO** statement was misformed, for example this is illegal

```
ON X PRINT
```

The word **ON** must be followed by a numeric variable 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 the assembler.

**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. e.g.

```
AUTO 100,0
```

**Syntax 220**

This error is generated by the cassette filing system. It indicates that a syntax error, such as an illegal **\*OPT** statement has occurred.

**String too 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 10. This can sometimes be caused by returning to a **FOR** statement without executing a **NEXT** statement. e.g.

```
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 sometimes be caused by returning to a **GOSUB** statement without executing a **RETURN** statement. e.g.

```
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. e.g.

```
10 REPEAT
20 GOTO 10
```

## Type mismatch 6

This error indicates that a number was expected and a string was offered or vice-versa. e.g.

```
10 A$=X
```

Num	Error message	Num	Error message
1	Out of range	30	Bad call
2	Byte	31	Arguments
3	Index	32	No FOR
4	Mistake	33	Can't match
5	Missing ,		FOR
6	Type mismatch	34	FOR variable
7	No FN	35	Too many FORs
8	\$ range	36	No TO
9	Missing "	37	Too many GOSUBs
10	Bad DIM	38	No GOSUB
11	DIM space	39	ON syntax
12	Not LOCAL	40	ON range
13	No PROC	41	No such line
14	Array	42	Out of DATA
15	Subscript	43	No REPEAT
16	Syntax error	44	Too many
17	Escape		REPEATs
18	Division by zero	45	Missing #
19	String too long	216	Data?
20	Too big	217	Header?
21	-ve root	218	Block?
22	Log range	219	File?
23	Accuracy lost	220	Syntax
24	Exp range	222	Channel
25	Bad MODE	223	EOF
26	No such variable	250	Key in use
27	Missing )	251	Bad key
28	Bad HEX	253	Bad string
29	No such FN/PROC	254	Bad command

# 47 Minimum abbreviations

This section 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 bracket because the token includes the bracket.

ABS	ABS	94	DRAW	DR.	DF
ACS	ACS	95	ELSE	EL.	8B
ADVAL	AD.	96	END	END	E0
AND	A.	80	ENDPROC	E.	E1
ASC	ASC	97	ENVELOPE	ENV.	E2
ASN	ASN	98	EOF	EOF	C5
ATN	ATN	99	EOR	EOR	82
AUTO	AU.	C6	ERL	ERL	9E
BGET	B.	9A	ERR	ERR	9F
BPUT	BP.	D5	ERROR	ERR.	85
CALL	CA.	D6	EVAL	EV.	A0
CHAIN	CH.	D7	EXP	EXP	A1
CHR\$	CHR.	BD	EXT	EXT	A2
CLEAR	CL.	D8	FALSE	FA.	A3
CLG	CLG	DA	FN	FN	A4
CLOSE	CLO.	D9	FOR	F.	E3
CLS	CLS	DB	GCOL	GC.	E6
COLOUR	C.	FB	GET	GET	A5
COS	COS	9B	GET\$	GE.	BE
COUNT	COU.	9C	GOSUB	GOS.	E4
DATA	D.	DC	GOTO	G.	E5
DEF	DEF	DD	HIMEM	H.	93
DEG	DEG	9D		(right)	
DELETE	DEL.	C7	HIMEM	H.	D3
DIM	DIM	DE		(left)	
DIV	DIV	81	IF	IF	E7

INKEY	INKEY	A6	PROC	PRO.	F2
INKEY\$	INK.	BF	PTR	PT.	8F
INPUT	I.	E8		(right)	
INSTR(	INS.	A7	PTR	PT.	CF
INT	INT	A8		(left)	
LEFT\$(	LE.	C0	RAD	RAD	B2
LEN	LEN	A9	READ	REA.	F3
LET	LET	E9	REM	REM	F4
LINE	LIN.	86	RENUMBER	REN.	CC
LIST	L.	C9	REPEAT	REP.	F5
LN	LN	AA	REPORT	REPO.	F6
LOAD	LO.	C8	RESTORE	RES.	F7
LOCAL	LOC.	EA	RETURN	R.	F8
LOG	LOG	AB	RIGHT\$(	RI.	C2
LOMEM	LOM.	92	RND	RND	B3
	(right)		RUN	RUN	F9
LOMEM	LOM.	D2	SAVE	SA.	CD
	(left)		SGN	SGN	B4
MID\$(	M.	C1	SIN	SIN	B5
MOD	MOD	83	SOUND	SO.	D4
MODE	MO.	EB	SPC	SPC	89
MOVE	MOV.	EC	SQR	SQR	B6
NEW	NEW	CA	STEP	S.	88
NEXT	N.	ED	STOP	STO.	FA
NOT	NOT	AC	STR\$	STR.	C3
OFF	OFF	87	STRING\$(	STRI.	C4
OLD	O.	CB	TAB(	TAB(	8A
ON	ON	EE	TAN	T.	B7
OPENIN	OP.	8E	THEN	TH.	8C
OPENOUT	OPENO.	AE	TIME	TI.	91
OPENUP	OPENUP	AD		(right)	
OR	OR	84	TIME	TI.	D1
OSCLI	OS.	FF		(left)	
PAGE	PA.	90	TO	TO	B8
	(right)		TRACE	TR.	FC
PAGE	PA.	D0	TRUE	TRUE	B9
	(left)		UNTIL	U.	FD
PI	PI	AF	USR	USR	BA
PLOT	PL.	F0	VAL	VAL	BB
POINT(	PO.	B0	VDU	V.	EF
POS	POS	B1	VPOS	VP.	BC
PRINT	P.	F1	WIDTH	W.	FE

































































































# 48 Appendix

	0	10	20	30	40	50	60	70	80	90	100	110	120
0	nothing	down	nothing	move cursor to 0,0									
1	next to printer	up	disable VDU	move cursor									
2	start printer	clear screen	select mode										
3	stop printer	start of line	reprogram charac's										
4	nothing	paged mode	nothing										
5	nothing	scroll mode	nothing										
6	enable VDU	nothing	nothing										
7	beep	nothing	nothing										back space & delete
8	back	nothing	nothing										nothing
9	forward	nothing	nothing										alpha red

### Teletext (MODE 7) Displayed Alphanumeric Characters

Each code produces a unique character. Thus VDU 78 or PRINT CHR\$(78) would display an N since column 70, row 8 shows an N.



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 * backgr'nd										
steady *	graphic yellow	new backgr'nd										
nothing	graphic blue	hold graphics										
nothing	graphic magenta	release * graphics										

\* every line starts with these options















































































	0	10	20	30	40	50	60	70	80	90	100	110	120
0	nothing	down	nothing	move cursor to 0,0									
1	next to printer	up	disable VDU	move cursor									
2	start printer	clear screen	select mode										
3	stop printer	start of line	reprogram charac's										
4	nothing	paged mode	nothing										
5	nothing	scroll mode	nothing										
6	enable VDU	nothing	nothing										
7	beep	nothing	nothing									back space & delete	
8	back	nothing	nothing									nothing	
9	forward	nothing	nothing									alpha red	

### Teletext (MODE 7) Displayed Graphic Characters


















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										
flash	graphic green	black * backgr'nd										
steady *	graphic yellow	new backgr'nd										
nothing	graphic blue	hold graphics										
nothing	graphic magenta	release * graphics										

\* every line starts with these options

	0	10	20	30	40	50	60	70	80	90	100
0	nothing	down	default logical colours	move text cursor to 0,0							
1	next to printer	up	disable VDU	move text cursor							
2	start printer	clear screen	select mode								
3	stop printer	start of line	reprogram charac's								
4	separate cursors	paged mode	define graphics area								
5	join cursors	scroll mode	plot								
6	enable VDU	clear graphics	default text/graphics areas								
7	beep	define text colour	nothing								
8	back	define graphics colour	define text area								
9	forward	define logical colours	define graphics origin								

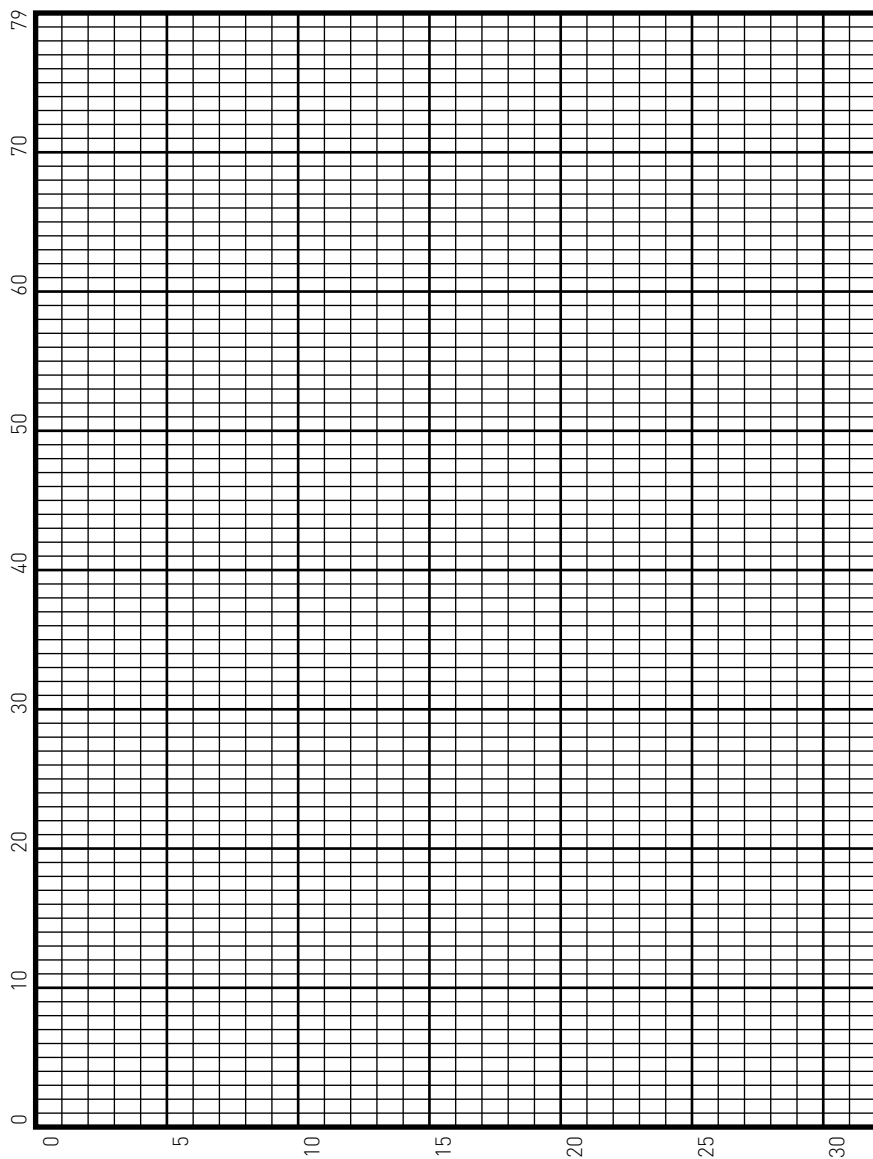
**ASCII (MODES 0 to 6) Displayed Character Set and Control Codes**  
Each displayed character consists of 8 rows of 8 dots.

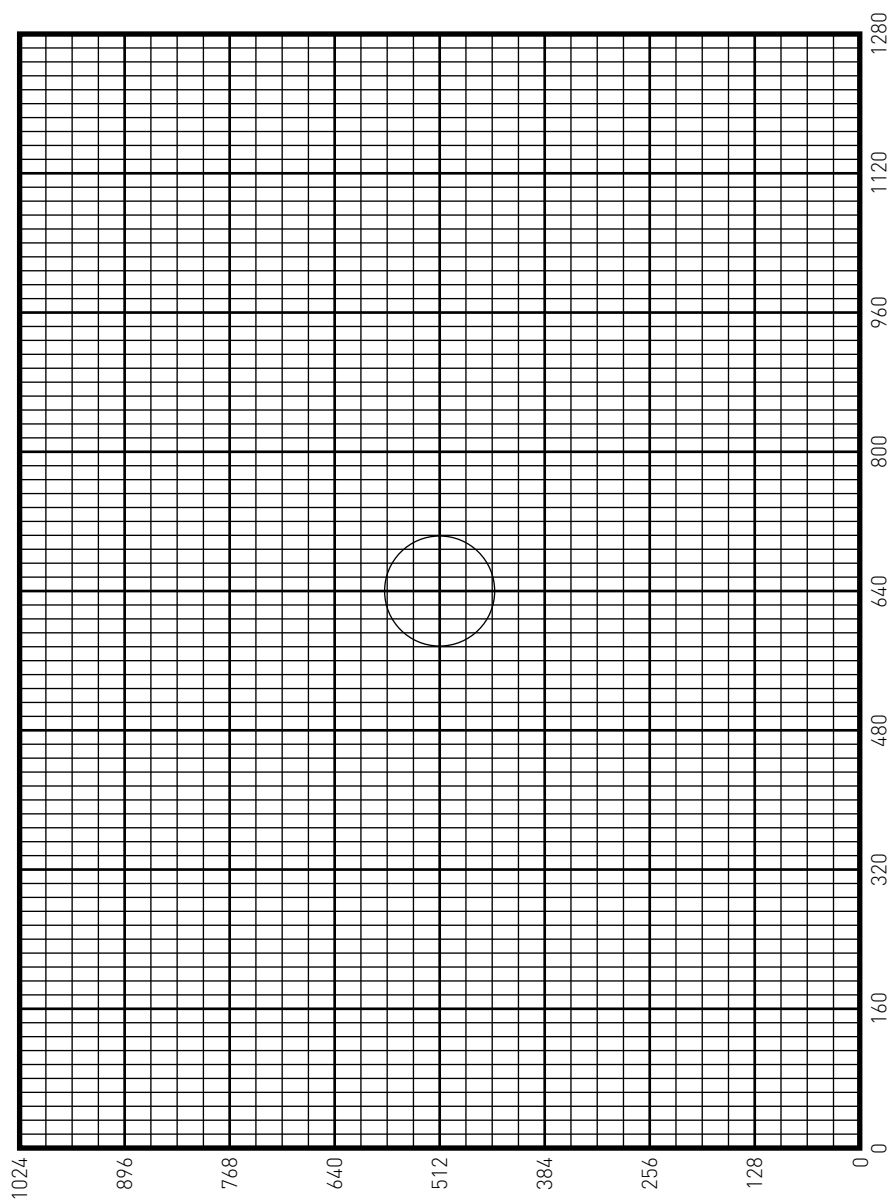
110	120	130	140	150	160	170	180	190	200	210	220	230	240	250
														
														
														
														
														
														
														
	back space & delete													
														
														

ALL CHARACTERS  
UNDEFINED INITIALLY

MSB LSB	8 0	9 1	A 2	B 3	C 4	D 5	E 6	F 7
0000 0	Nothing	Clear graphics area		0	@	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 colours	\$	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 0,0	.	>	N	^	n	~
1111 F	Paged mode off	Move text cursor to X,Y	/	?	O	—	o	Backspace and delete

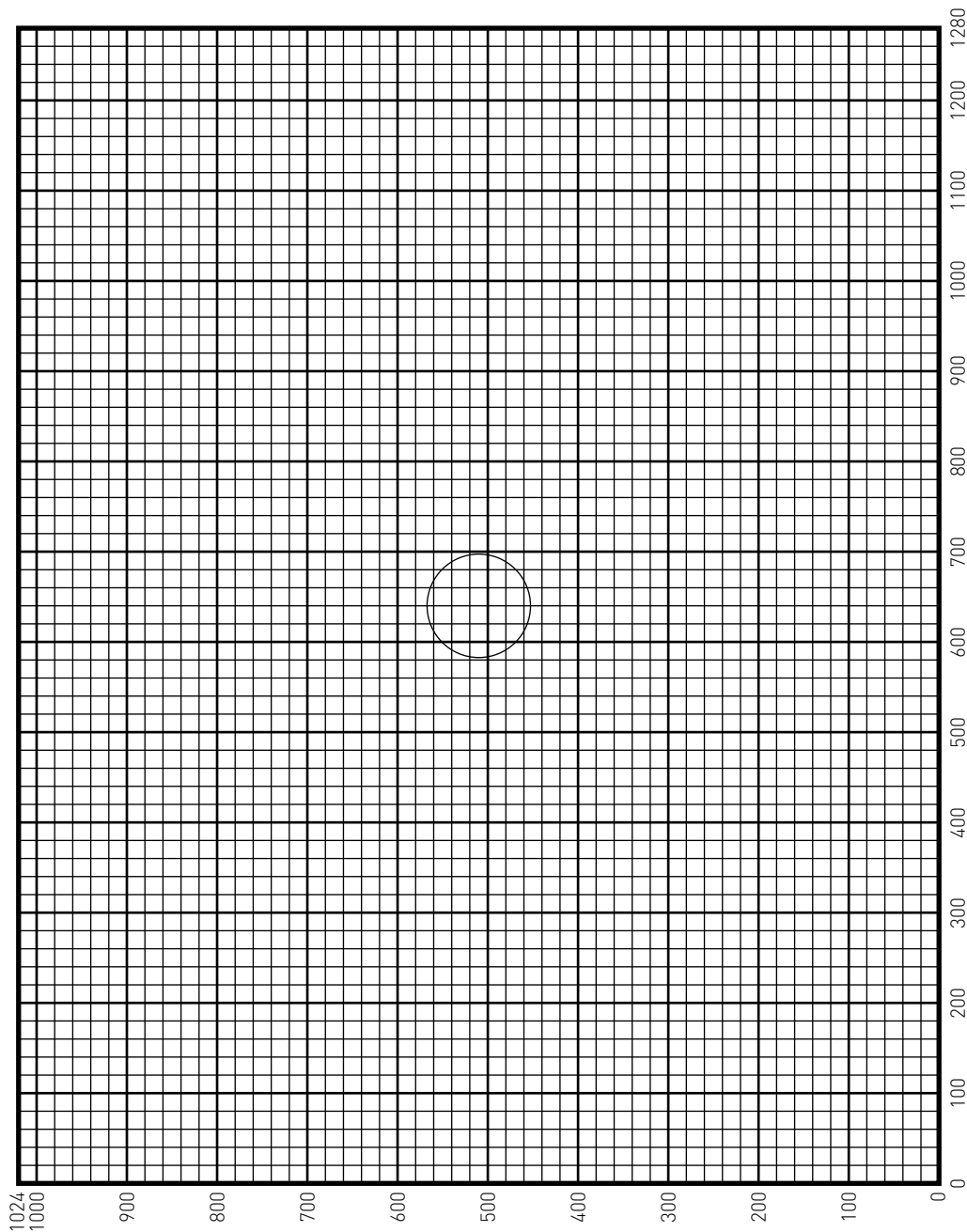
## Hexadecimal ASCII Codes



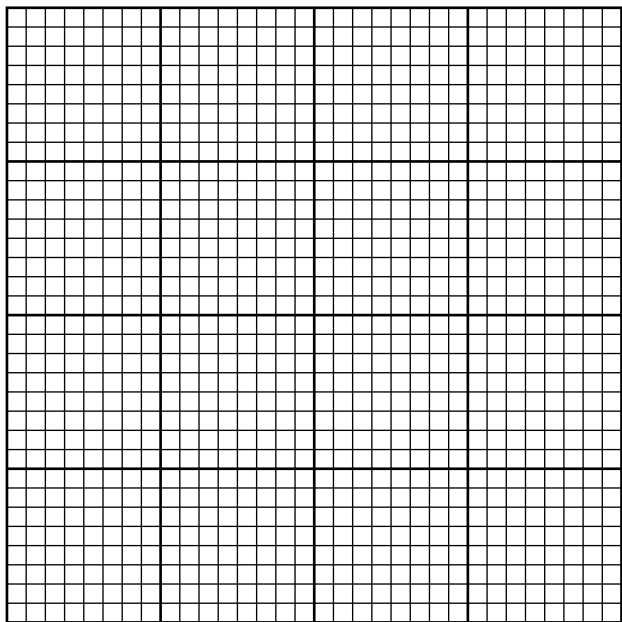


Graphics Planning Sheet 1 (grid related to character positions)



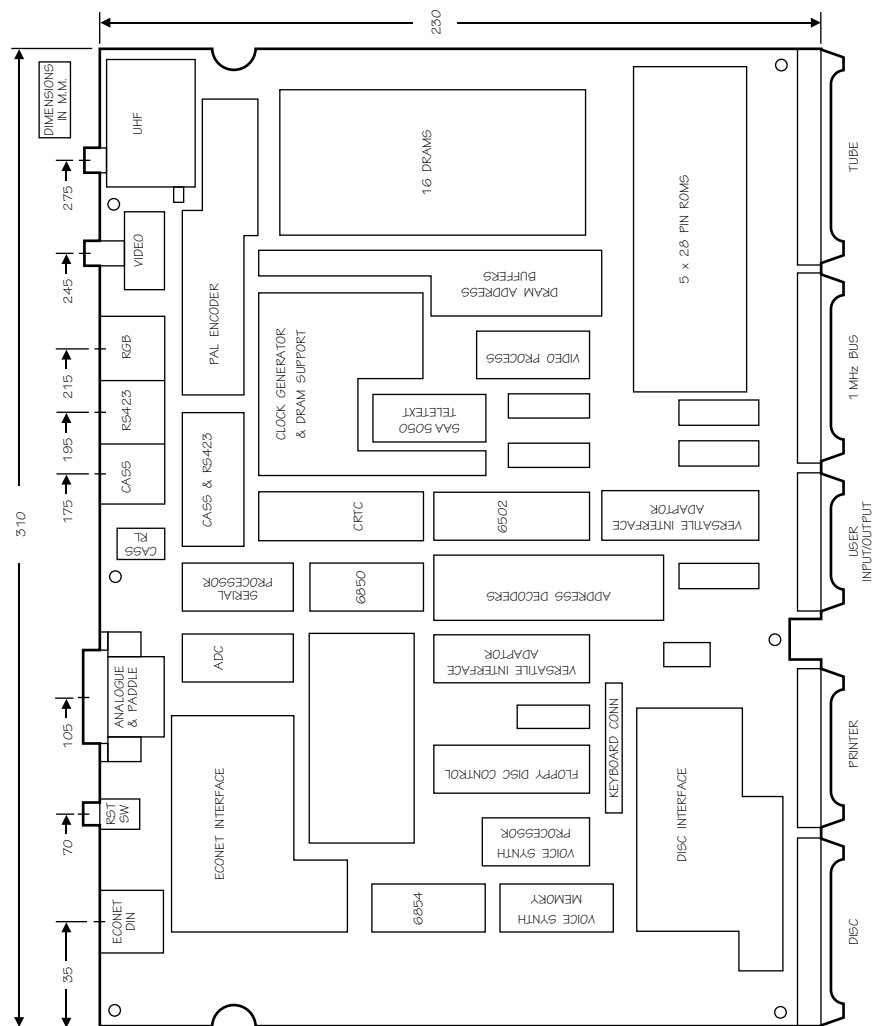


128	
64	
32	
16	
8	
4	
2	
1	

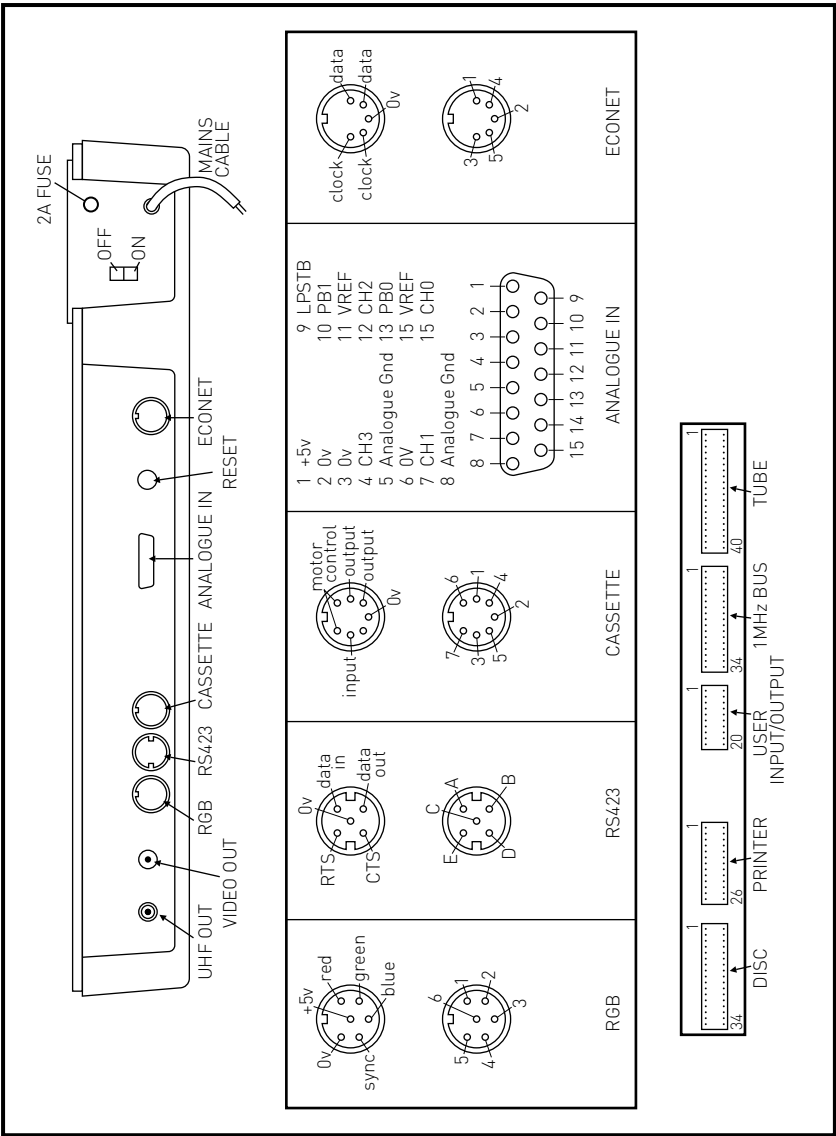


**User Defined Character Planning Sheet**  
MODES 0 to 6 (see page 172)

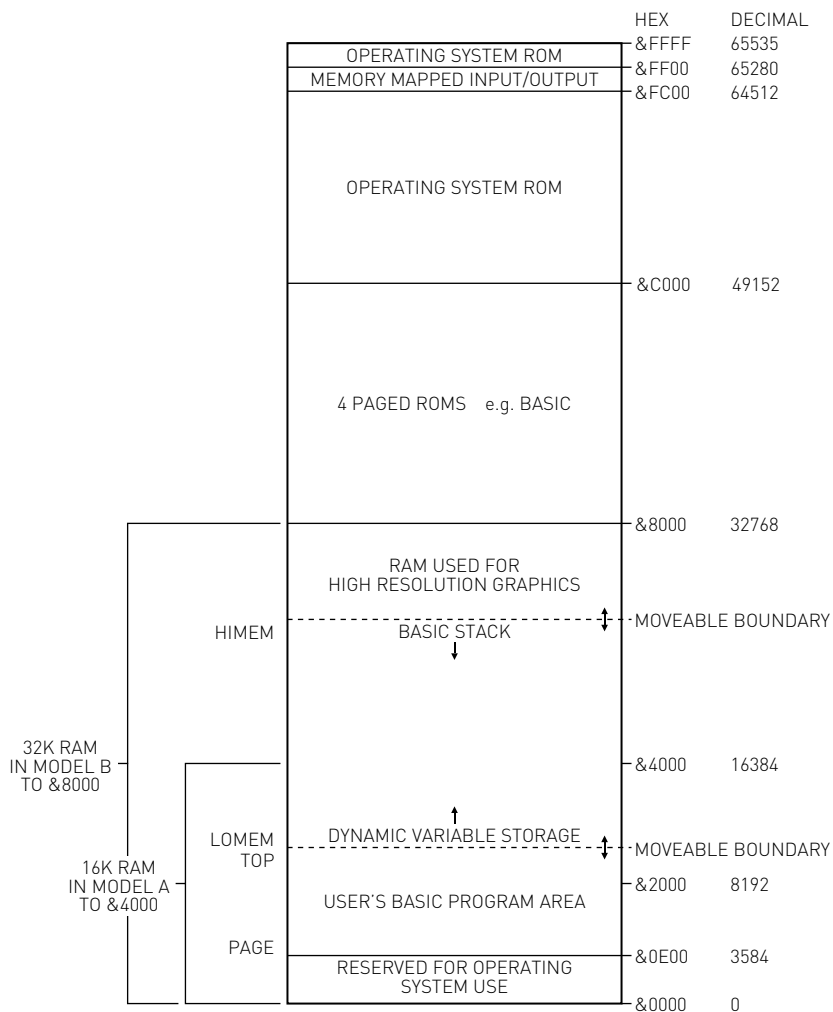




Printed circuit board layout for the BBC Microcomputer



External connections



## Memory map

space for operating system resident routines	&0E00	3584
user defined character definitions	&0D00	3328
user defined function key definitions	&0C00	3072
various buffers	&0B00	2816
various buffers	&0A00	2560
misc. workspace	&0900	2304
language ROM workspace	&0800	2048
misc. workspace	&0400	1024
operating system workspace	&0300	768
6502 stack	&0200	512
zero page	&0100	256
	&0000	0

### Memory map (detail)

## Memory Map Assignments

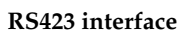
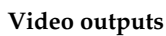
FF00 - FFFF	Operating System ROM
FE00 - FEFF	Internal memory mapped input/output (SHEILA)
FD00 - FDFE	External memory mapped input/output (JIM)
FC00 - FCFE	External memory mapped input/output (FRED)
C000 - FBFF	Operating System ROM
8000 - BFFF	One or more languages ROMs (e.g. BASIC, PASCAL)
4000 - 7FFF	Optional RAM on Model B
0000 - 3FFF	always RAM
E00	Default setting of <b>PAGE</b>
D80 - DFF	allocated to machine operating system
D00 - D7F	Used by NMI routines (eg by Disc or Econet filing systems)
C00 - CFF	User defined character definitions
B00 - BFF	User defined function key definitions
A00 - AFF	RS423 receive, and cassette workspace
900 - 9FF	RS423 transmit, cassette, sound and speech workspace
800 - 8FF	Miscellaneous workspace
400 - 7FF	Language ROM workspace
300 - 3FF	Miscellaneous workspace
200 - 2FF	Operating system workspace and indirection vectors
100 - 1FF	6502 stack
000 - 0FF	Zero page

### Zero Page

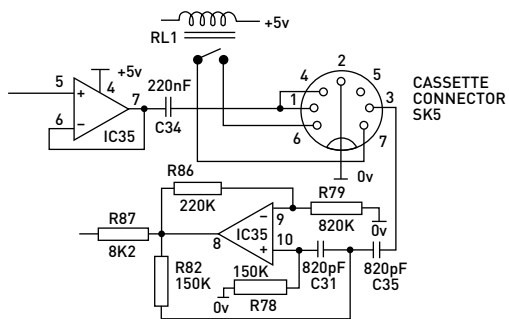
FF	The top bit is set during an <b>ESCAPE</b> condition
FD - FE	Address following detected BRK instruction
FC	User IRQ routine save slot for register A
D0 to FB	allocated to machine operating system
B0 to CF	allocated to current filing system
90 to AF	allocated to machine operating system
70 to 8F	free for user routines
00 to 6F	BASIC language



### Printer, User I/O and 1MHz Bus circuits







**Cassette interface**

Decimal	Hex	CTRL	ASCII abbrev.	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 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 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	5	define logical colour
20	14	T	DC4	0	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	re-program display character
24	18	X	CAN	8	define graphics window
25	19	Y	EM	5	<b>PL</b> <b>O</b> <b>T</b> <b>K</b> , <b>x</b> , <b>y</b>
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

## VDU code summary

# 6502 instruction set

INSTRUCTIONS		IMMEDIATE		ABSOLUTE		ZERO PAGE		ACCUM		IMPLIED		INDX		INDY		ZPAGEX		ABSY		RELATIVE		INDIRECT		ZPAGEY		PROCESSOR STATUS CODES		MNEM.							
MNE/MONIC	OPERATOR	OP n #	n #	OP n #	OP n #	OP n #	OP n #	OP n #	OP n #	OP n #	OP n #	OP n #	OP n #	OP n #	OP n #	OP n #	OP n #	OP n #	OP n #	OP n #	OP n #	OP n #	OP n #	OP n #	OP n #	N	V	B	D	I	Z	C			
ADC	A ← M ← C → A	(4) 0	1	69	2	2	AD	4	3	65	3	2														N	V	.	.	.	.	Z	C	ADC	
AND	A ← M → A	(1)	29	2	2	2D	4	3	25	3	2															N	.	.	.	.	.	Z	C	AND	
ASL	C ← 7 0 ← 0						0E	6	3	06	5	2														N	.	.	.	.	.	Z	C	ASL	
BCC	BRANCHON C=0	(2)																								.	.	.	.	.	.	.	.	BCC	
BCS	BRANCHON C=1	(2)																								.	.	.	.	.	.	.	.	BCS	
BEQ	BRANCHON Z=1	(2)																								.	.	.	.	.	.	.	.	BEQ	
BIT	A ← M						2C	4	3	24	3	2														M	.	.	.	.	.	.	.	BIT	
BMI	BRANCHON N=1	(2)																								.	.	.	.	.	.	.	.	BMI	
BNE	BRANCHON Z=0	(2)																								.	.	.	.	.	.	.	.	BNE	
BPL	BRANCHON N=0	(2)																								.	.	.	.	.	.	.	.	BPL	
BRK	BREAK										00	7	1													.	.	.	.	.	.	.	.	BRK	
BVC	BRANCHON V=0	(2)																								.	.	.	.	.	.	.	.	BVC	
BVS	BRANCHON V=1	(2)																								.	.	.	.	.	.	.	.	BVS	
CLC	0 → C										18	2	1													.	.	.	.	.	.	.	.	CLC	
CLD	0 → D										D8	2	1													.	.	.	.	.	.	.	.	CLD	
CLI	0 → I										58	2	1													.	.	.	.	.	.	.	.	CLI	
CLV	0 → V										B8	2	1													.	.	.	.	.	.	.	.	CLV	
CMP	A ← M	C9	2	2	CD	4	3	C5	3	2																N	.	.	.	.	.	.	.	CMP	
CPX	X ← M	B0	2	2	BC	4	3	E4	3	2																N	.	.	.	.	.	.	.	CPX	
CPY	Y ← M	C0	2	2	CC	4	3	C4	3	2																N	.	.	.	.	.	.	.	CPY	
DEC	M ← 1 → M																									N	.	.	.	.	.	.	.	.	DEC
DEX	X ← 1 → X										CA	2	1													N	.	.	.	.	.	.	.	DEX	
DEY	Y ← 1 → Y										88	2	1													N	.	.	.	.	.	.	.	DEY	
EOR	A ← M → A	(1)	49	2	2	4D	4	3	45	3	2															N	.	.	.	.	.	.	.	EOR	
INC	M ← 1 → M										EE	6	3	E6	5	2										N	.	.	.	.	.	.	.	INC	
INX	X ← 1 → X										B8	2	1													N	.	.	.	.	.	.	.	INX	
INY	Y ← 1 → Y										C8	2	1													N	.	.	.	.	.	.	.	INY	
JMP	JUMPTO NEW LOC										4C	3	3													.	.	.	.	.	.	.	.	JMP	
JSR	JUMPSUB										20	6	3													.	.	.	.	.	.	.	.	JSR	
LDA	M → A	(1)	A9	2	2	AD	4	3	A5	3	2															N	.	.	.	.	.	.	.	LDA	



## \*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	Flush 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 synchronization
20	14	Explode soft character RAM allocation
21	15	Flush selected buffer
* 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
136	88	Reserved
* 137	89	Turn cassette motor ON/OFF
138	8A	Insert character into keyboard buffer



<b>Decimal</b>	<b>Hex</b>	<b>Function</b>
* 139	8B	Set file options
* 140	8C	Select cassette file system and set speed
141	8D	Reserved
142	8E	Reserved
143	8F	Reserved
* 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
218	DA	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

Routine		Vector		Summary of function
Name	Address	Name	Address	
		UPTV	222	User print routine
		EVNTV	220	Event interrupt
		FSCV	21E	File system control entry
OSFIND	FFCE	FINDV	21C	Open or close a file
OSGBPB	FFD1	GBPBV	21A	Load or save a block of memory to a 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

# OPENUP open 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#
```

# 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$Y
40 OSCLI A$
```

## Example

```
10 FN$="XYZ" : REM FILE NAME
20 START%=&4000 : REM START OF CODE
30 END%=&6000 : REM END OF CODE
40 EXEC%=&5000 : REM EXEC ADDRESS
300 OSCLI "SAVE "+FN$+" "+STR$~(
START%)+ " "+STR$~(END%)+ " "+STR$~(
EXEC%)
```

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\$**



# Index

## A

Abbreviations for keywords 483  
ABS 200  
Acknowledge escape conditions 429  
ACS 201  
Accuracy of calculations 65  
Actual colour numbers 165  
Addressing modes 449  
ADSR envelope 185, 244, 463  
ADVAL 202, 426, 429, 467  
Aligning columns when printing 72  
Amplitude envelope 184, 244  
Analogue input connections 467  
Analogue to digital converter 202, 426, 429, 467  
AND 205  
Animation 169  
Appending programs 402  
Application note 436  
Arc-cosine 201  
Arc-sine 209  
Arc-tangent 210  
Arrays 120, 236  
ASC 65, 207  
ASCII 64, 207, 490  
ASN 209  
Assembly language CALL 214, 446  
Assembly language DIM 236, 444  
Assembly language examples 420  
Assembly language introduction 442  
Assembly language monitor 411  
Assembly language OPT 314  
Assembly language USR 371, 445  
ATN 210  
Attack rate 184, 244, 461  
AUTO 53, 211  
Automatic line numbers 211  
Autopaging 38, 381  
Auto repeat of keys 9, 425

## B

Background colours 55, 60, 162  
Bad program 475  
Base value of function keys 439  
Baud rate selection on cassette 400, 434  
Baud rate selection on RS423 424  
BGET# 212  
Bitwise AND 205  
Boolean types 259  
BPUT# 213  
Break key 17, 142  
BRIAN 48  
BRK 464  
Buffer flushing  
  all 426  
  input 426, 427  
  keyboard 426  
  sound 427  
Buffer get character 430, 435

Buffer insert character 429, 433  
Buffer status 429, 465

## C

Calendar program 131  
CALL 214, 446  
Cambridge ring 473  
CAPS LOCK key 15  
Cartridge socket 19  
Cartridge ROM file system 472  
Cassette file internal format 328, 399  
Cassette file system 188, 390  
Cassette leads 12  
Cassette loading 292  
Cassette motor control 390  
Cassette motor relay on/off 416, 433  
Cassette recordings 34, 390  
Catalogue 391  
Catalogue of cassette tape 391  
Centronics printer 404  
CHAIN 36, 216, 292  
Channels when using files 191  
Character set 486  
Character – user defined 172, 386, 429  
CHR\$ 65, 217  
Circuit board layout 498  
Circuit diagrams 503  
CLEAR 218  
Clear graphics window 60, 220  
Clearing the screen 220, 381  
Clearing text window 60, 221  
CLG 60, 220, 381  
CLI 463  
Clock 84, 364, 459  
Clock program 131  
CLOSE# 219  
CLS 60, 221, 381  
COLOUR 55, 60, 222, 262  
Commands 21  
Command line interpreter 463  
Command mode 29  
Comments in assembly language 449  
Comments in BASIC programs 53, 334  
Common variables 66  
Concatenation of strings 64, 361  
Connectors 499  
Contents of memory 411  
Control codes 378, 490  
CONTROL key 18  
Co-ordinates on screen 56  
COPY key 30, 82, 422  
Correcting errors 29  
COS 225  
Cosine 225  
COUNT 226  
CP/M 473  
CRTC register access 385, 437  
CTRL U 80  
Cursor control codes 75, 77, 382

Cursor editing 30  
Cursor off 77  
Cursor position 323, 375, 432

## D

Data 123, 126, 227  
Data files on cassette 395, 396  
Data Logging 395  
Date 131  
Decimal places 70, 325  
Decimal point 21  
DEF 230  
Defining characters 174, 384, 427  
DEG 234  
Degrees from radians 234  
Delete current entry 80, 383  
Delete key 17  
Delete whole line 31, 53, 235  
Demonstration programs  
  Age 78  
  BL and Lotus 128  
  Brian 48  
  Call 446  
  Div and Mod 130  
  Double height teletext 50  
  Draw 83  
  Drinks 190  
  Fourpnt 44  
  FX Demo 420, 433  
  Geography quiz 228  
  GOTO 32  
  Hand Mouth Ear 266  
  Hangman 138  
  Hanoi 329  
  Hours, Mins, Secs 131  
  Hypno 105  
  H2 74  
  Leap years 134  
  Lunar Landing 176  
  Man 171  
  Month 126  
  Monthly 40  
  People and arrays 124  
  Persian 46  
  Polygon 39  
  Quadrat 42  
  React 102  
  Read screen character 432  
  Reverse string 136  
  Rocket 173  
  Role 75  
  Sine 49  
  Sine in teletext 157  
  Sqr root 47  
  Stars and stripes 93  
  Sums in 15 seconds 87  
  Tartan 45  
  Telephone book 193  
  Temperature 115  
  Too late 81  
  Windows 61

## E

Econet file system 400, 472  
Editing a line 29  
Editing keys 30, 82, 422  
Editing key produces codes 82, 422  
Effects 418  
ELSE 241  
Enable screen output 380  
END 242, 395, 429  
End of file 251, 395, 429  
ENDPROC 245  
Entry point in assembler 450  
ENVELOPE 183, 244, 461  
EOF# 249, 397, 429  
EOR 250  
Erasing the screen 60, 220, 381  
ERL 148, 251  
ERR 147, 252  
Error codes 147, 252, 482  
Error handling 147, 309, 338, 397, 434  
Error handling in assembler 314, 448  
Error line 148, 251  
Error message 474  
Error numbers 397, 482  
Errors, correcting 29  
Escape acknowledge 429  
Escape detected (assembler) 466  
ESCAPE error code 476  
Escape key 17, 428  
Escape reset 428  
Escape set 428  
EVAL 253  
Evaluate a string 253  
Event enable 425  
Event disable 426  
Event handling 464  
Exclusive OR in BASIC 250  
EXP 255  
Expansion bus 435, 473  
Expansion options 471  
Exponent 21  
EXT# 256

## F

FALSE 89, 100, 257, 369  
Fault handling 464  
Fields 67  
Field Sync 427, 465  
Field width 67, 70  
Filenames 397  
Files 188, 451  
File pointer 330  
File system 400  
Filling an area with colour 162  
Fire button on games paddle 203, 429, 467  
Flashing colours 165, 424  
Flash rate selection 424  
Floppy disc interface 400, 471  
Flush keyboard buffer 426, 427  
Flush in put buffer 426, 427  
Flush VDU queue 438  
FN 259



FOR...NEXT 91, 94, 260, 305, 356  
Foreground colours 55, 60, 160  
FOURPNT 44  
Free space left 414  
FRED 436  
Functions 110, 230, 259  
f0 function keys 22, 141, 427, 439  
FX call summary 418

## G

Games paddles 202, 429, 467  
GCOL 56, 167, 262  
Geography quiz 228  
GET 263  
Get character from buffer 81, 273, 430, 435  
GET\$ 264  
Global variables 107  
GOSUB 113, 265  
GOTO 117, 268  
Graphics 55, 160  
Graphics origin 388  
Graphics planning sheet 494  
Graphics windows 57, 385

## H

Hard reset 142  
Hexadecimal 71  
High order address 431  
HIMEM 270, 414  
HYPNO 105

## I

IF...THEN...ELSE 98, 241, 272, 365  
Indirection operators 409, 450  
INKEY 81, 275, 430, 435  
INKEY\$ 267  
INPUT 78, 277  
INPUT# 279  
INPUT LINE 278  
INPUT line 461  
Input/Output devices 435, 469  
Input stream selection 421  
INSTR 136, 280  
Instruction set for 6502 507  
INT 282  
Integer arithmetic 130, 238, 299  
Integer variables 65, 410  
Internal file format 328, 399  
Internal format in memory  
  of BASIC 484  
  of variables 63, 66  
Interrupts 440, 441, 466  
Interval timer 460, 466  
Inverse colour 167  
IRQ handling 440, 441, 466

## J

Jim 436  
Joysticks 202, 429, 467

## K

Keyboard 15

Keyboard auto repeat 9, 425  
Keyboard testing for BASIC 273  
Keyword definitions 197  
Keywords – details 197  
Keywords – summary 484

## L

Leads for cassette 12  
Leap year calculation 134  
LEFT\$ 135, 283  
Length of a file 256  
Length of a program 414  
Length of a string 136, 285  
LET 25, 287  
Line numbers 28  
LIST 288  
LISTO 94, 290  
List options 94, 290  
List processing 413  
LN 291  
LOAD 292  
LOADGO (CHAIN) 36, 216, 394  
Loading machine code 393  
Loading programs 34, 393, 394  
LOCAL variables 107, 294  
LOG 295  
Logarithm 295  
Logical colours 164  
LOMEM 296, 414  
Loops 86, 414  
Lunar lander game 177

## M

Machine code 442  
Machine operating system 452  
Man shaped character 171  
Mantissa 63, 66  
Memory expansion 473  
Memory maps 500, 501, 502  
Memory pointers 270, 290, 317, 366, 414, 431  
Memory – saving 195  
Merging programs 402  
MID\$ 135, 298  
Mistake 478  
MOVE 56, 303  
MODE 55, 160, 222, 301  
MODE7 150  
Monitor, colour 471  
Monitor lead 7  
MONTHLY 40  
Motor on/off 433, 463  
MOVE 56, 303  
Multiple statement lines 54, 98  
Music 11  
Musical notes 182

## N

NETwork file system 400, 472  
NEW 304, 307  
NEXT 91, 94, 260, 305  
NMI 467  
Noise generator 348

NOT 306  
Note synchronisation 187  
Number to string conversion 137, 358  
Numeric accuracy 65  
Numeric range 65  
Numeric variables 24, 65

## O

OLD 307  
ON ERROR 117, 308  
ON GOSUB 308  
ON GOTO 117, 308  
OPENIN 311  
OPENOUT 313  
OPENUP 513  
Opening file for input 311  
Opening file for output 313  
Operating system call summary 452  
Operating system statements 416  
Operator precedence 144  
OPT 314, 447  
OR 316  
Origin move 388  
OSARGS 454  
OSASCI 457  
OSBGET 453  
OSBPUT 453  
OSBYTE calls 418  
OSCLI 463, 514  
OSFILE 454  
OSFIND 451  
OSGBPB 453  
OSNEWL 457  
OSRDCH 456  
OSWORD 458  
OSWRCH 408, 457  
Output stream select 422

## P

PAGE 317, 414  
Page mode 38, 381  
Panic button 29  
Parameters 105  
Parameter block in CALL 214, 446  
PEEK 409  
PERSIAN 46  
PI 318  
Pitch envelope 183, 244  
Pling indirection operator 409  
PLOT statement 319  
PLOT a point 169  
POINT 322  
Pointers to memory 366, 296, 270, 317, 414  
POKE 409  
POLYGON 39  
POS 323  
Precedence of operators 144  
PRESTEL file system 401, 472  
PRINT 324  
PRINT# 328  
Printer on/off 288, 407, 422  
Printer  
    choosing 471

connections 404  
drivers 408  
on/off 288, 407, 422, 423  
parallel 404  
serial 404

## Q

Quadrat 42  
Query indirection operator 409  
Qume printer 407

## R

RAD 331  
Radians from degrees 331  
RAM 414  
Random numbers 84, 342  
Range numeric 65  
REACT 102  
READ 123, 126, 332  
Read key 263, 81, 273, 430  
Read screen character 432  
Read screen point 322, 462  
Real variables 464  
Recording programs 34  
Red keys 22, 141, 427, 439  
Relay on/off 433, 437  
REM 53, 334  
Remarks in assembly language 449  
Remarks in programs 53, 334  
Remote control tape recorder 437, 463  
RENUMBER 32, 54, 335  
REPEAT...UNTIL 87, 337  
Report error 338  
Reserved words 484  
RESET 142  
Resident integer variables 66  
RESTORE 129, 339  
Return key 9, 17  
Return statement 340  
RIGHT\$ 135, 341  
RND 84, 342  
Rocket graphics shape 174  
ROM file system 401, 472  
RS232C printers 406  
RS423 as input 421, 435  
RS423 connections 406  
RUN 343

## S

SAVE 344  
Saving  
    a section of memory 392  
    BASIC programs 34, 344, 391  
    data 330  
    machine code 392  
    memory space 194, 359  
    single character 215  
Save format 330, 399  
Screen editor 29  
Screen size 55  
Scroll mode 38, 381  
Second processor 473

Sequential access files 188, 451  
 Serial port 406, 424, 441  
 Serial printer connections 406  
 Serial ULA bit meanings 437  
 SHEILA 436  
 SHIFT key 15  
 SHIFT lock 17  
 Sign of a number 345  
 Significant figures 65  
 SIN 346  
 SINE program 49, 157  
 SGN 345  
 Sockets on computer 499  
 Soft reset 142  
 SOUND 180, 347, 461  
 Spaces – printing on screen 360  
 SPC 354  
 Speeding up programs 195  
 SQR ROOT 335  
 SQR 47  
 Squares in graphics 161  
 Statements 21  
 Star commands 400, 416  
 Stars and Stripes 93  
 STEP 356  
 STOP 357  
 STR\$ 135, 358  
 STRING\$ 137, 359  
 String concatenation 63, 359  
 String functions 110, 135  
 String indirection operator 409  
 String-length of 136, 285  
 String-multiple copies of 137, 359  
 String-searching for one in another 136, 280  
 String-to-number conversion 137, 372  
 String variables 63, 66, 135, 359  
 Structures in BASIC 86, 413  
 Sub-routines in BASIC 113, 267  
 Synchronisation of sounds 158  
 Syntax explanation 198

## T

TAB 72, 360  
 Tabulation 72  
 TAN 362  
 Tangent 362  
 Tape file system 190, 390, 400  
 TARTAN 44  
 Telesoftware 472  
 Telephone book program 193  
 Teletext 50, 150, 439, 471  
 Teletext character set 18  
 Teletext control codes 50, 150, 439, 486  
 Teletext file system 400  
 Temperature conversion program 115  
 Text planning sheets 493  
 Text windows 58  
 THEN 363  
 TIME 84, 86, 364  
 Tokens 484  
 TOP 366, 416  
 TRACE 367

Triangles in graphics 162

TRUE 89, 100, 369

TUBE® 473

Tuning a TV 7

Types of variables 65

## U

Unplot a point 169

UNTIL 87, 370

User defined characters 171, 386, 427

User defined function keys 22, 141, 439

User input/output port address 437

User port 410

User supplied printer driver 408

USR 371, 445

## V

VAL 372

Variables 24, 62, 120

VDU 74, 373, 377

VDU queue flush 440, 373

VDU summary 378

VDU5 50, 74, 379

VDU19 164, 223, 382

Version number of operating system 421

VIA user port address 437

Voice synthesis unit 472

Volume settings 12, 390

VPOS 375

V24 port 408

## W

Wait for field sync 427

Welcome cassette 12

Whole number arithmetic 130, 238, 299

WIDTH 376

Windows 57, 381, 385, 387

## X

XY cursor addressing 73

## Z

Z80 473

1MHz expansion bus 436, 473

6522 addresses 437

16032 473

+ addition 21

+ concatenation of strings 64, 359

\* multiplication 21

\* star commands 400, 416

\*TAPE 400, 434

\*TV 23, 435

\*EXEC 394, 402

\*MOTOR 417, 433

\*OPT 397, 434

\*FX 418

\*RUN 392

\*CAT 391

\*KEY 141, 416, 439

\*LOAD 393

*SAVE	392	^	exponentiation	21
*FX3	408, 422	()	brackets	144
*FX4	82, 422	[]	square brackets	443
*FX5	407, 423	<		144
*FX6	407, 424	<=		144
*FX7	408, 424	=		144
		>		144
:	multiple statement	>=		144
;	in PRINT	"	quotation marks	324
;	in VDU	'	apostrophe	324
?	Indirection operator	←		30, 82, 422
!	Indirection operator	→		30, 82, 422
\$	Indirection operator	↑		30, 82, 422
	for string	↓		30, 82, 422
~	PRINT in hex		vertical bar	18, 22, 23
&	Hex number	!	vertical bar	18, 22, 23
@%	PRINT format	½		18
#	immediate	¼		18
/	division	¾		18
\	comment in assembler			



