acorn electron

ELECTRON

User Guide



acorn electron microcomputer

acorn electron

User Guide



The Electron User Guide

Part no 4Ø5ØØØ
Issue no 1
Date June 1983

CARE AND MAINTENANCE

Exposure
Like all electronic equipment, the Electron should not be exposed to direct sunlight for long periods.

Servicing
All servicing should be done through an authorised dealer. There are no serviceable parts inside either the Electron or the mains adapter, and opening either case will void the warranty.

The mains adapter
Note that your Electron Microcomputer does not have a separate on/off switch, so to switch it off unplug the mains adapter.

To switch the computer off briefly, you can remove the small plug on the right hand side of the case. However, the mains adapter should not be left plugged into the domestic 13A socket for long periods if the Electron is not being used.

There is a thermal fuse built into the mains adapter and if this blows, you will need to replace the adapter (please contact your local dealer). The Electron Microcomputer contains its own short circuit protection, so adapter failure is unlikely.

UNDER NO CIRCUMSTANCES should the mains adapter be replaced by a normal mains plug. The mains adapter contains a transformer which reduces the mains to a safe low voltage for input to the Electron.

AUTHOR'S NOTE: Wherever the letters BBC are mentioned in this book, they refer to the British Broadcasting Corporation.

© Copyright Acorn Computers Limited 1983

Neither the whole or any part of the information contained in, or the product described in, this manual may be adapted or reproduced in any material form except with the prior written approval of Acorn Computers Limited (Acorn Computers).

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

All correspondence should be addressed to:

Technical Enquiries
Acorn Computers Limited
Fulbourn Road
Cherry Hinton
Cambridge CB1 4JN

All maintenance and service on the product must be carried out by Acorn Computers' authorised dealers. Acorn Computers can accept no liability whatsoever for any loss or damage caused by service or maintenance by unauthorised personnel. This manual is intended only to assist the reader in the use of the product, and therefore Acorn Computers shall not be liable for any loss or damage whatsoever arising from the use of any information or particulars in, or any error or omission in, this manual, or any incorrect use of the product.

First published 1983
Published by Acorn Computers Limited
Typeset by Bateman Typesetters, Cambridge

Contents

1 What is a computer? 1

2 Getting started 2

Checklist of items 2
Additional items 2
Connecting the Electron to your television set 2
Connecting the Electron to the mains 4
Tuning the TV to the Electron 4
Push-button tuning 5
Single tuning knob 5
Connecting the Electron to a monitor 5
Monochrome monitor 5
Colour (RGB) monitor 5
Now try something 5

3 Using a cassette recorder 7

Introduction 7
Connecting a cassette recorder 7
Motor control 8

4 The Introductory Cassette 9

Adjusting the volume control and loading the first program 9
Loading each program from the cassette 14

5 How to use the keyboard 16

Introduction 16
Choosing the keyboard characters 18
SHIFT and CAPS LK 18
SHIFT 18
FUNC 19
The arrow keys and the COPY key 19
What the arrow and COPY functions do 19
Summary 20

6 Introducing commands and programs 21

What is hexadecimal? 23

7 Editing programs 24

Introduction 24
Listing the program 25
Editing programs 26
Editing with the arrow keys and the COPY key 26
Deleting lines from your program 28
Inserting new lines into your program 29
Renumbering the program 30
Getting the computer to number each program line 31
Putting notes into your programs 32
Retrieving a program and starting a new one 32
Listing long programs 33

8 Trying out some programs 34

Introduction 34
PERSIAN 34
POLYGON 35
DRAW 36

9 Recording programs on cassette 38

Saving (recording) a program on cassette 38
Checking a recording 39
Loading a program from cassette 39
Cataloguing the tape 41
What the numbers mean 41
Escape 41

10 The FUNC key and BASIC keywords 42

11 Variables and expressions 44

What is a variable? 44
Real variables 44
Operators and expressions 45
Rules for variable names 46
Integer variables 47

A% to Z% 47
Real versus integer variables 48
DIV and MOD 48
The TIME integer variable 49
String variables 49
Commands operating on strings 51
LEN 51
Linking strings 52
LEFT$, RIGHT$, MID$ 52
VAL, EVAL, STR$ 53
INSTR 53
STRING$ 54
Comparison table of variables 54

12 Operator precedence 55

13 Arrays 56

14 READ . . . DATA . . . RESTORE 58

15 PRINT formatting and INPUT 59

PRINT formatting 59
INPUT 64

16 Conditional and loop instructions 68

The FOR . . . NEXT loop 68
The REPEAT . . . UNTIL loop 72
IF . . . THEN . . . ELSE 73

17 Procedures 77

Using parameters in procedures 81

18 GOTO and GOSUB 84

GOTO 84
GOSUB . . . RETURN 84
ON . . . GOTO, ON . . . GOSUB 86

19 Functions 87

2Ø Graphics 89

Introduction 89
Modes – what are they and why? 89
Writing text 9Ø
The COLOUR command and text windows 9Ø
Addresses on the text screen 91
Text windows 92
Defining your own characters 93
Graphics 95
Introduction 95
The graphics coordinate system 96
The GCOL command 97
The PLOT command 99
Advanced graphics 99
Triangle plotting 99
Sideways filling on background colour 99
Filling right 1ØØ
The VDU command 1ØØ
Graphics windows 1Ø1
The graphics origin 1Ø1
Plotting characters 1Ø2
The palette 1Ø2

21 VDU codes 1Ø4

Introduction 1Ø4
Detailed description 1Ø4

22 Making sounds 116

Introduction 116
The SOUND command 116
The Q parameter 117
The A parameter 117
The P parameter 117
The D parameter 118
Using the SOUND command in a program 118
ENVELOPE 12Ø
The ENVELOPE command 121
Constructing an ENVELOPE 123

Additional SOUND features 125
Example SOUND and ENVELOPE programs 125

23 Address pointers, indirection operators 127

The Electron's memory 127
Indirection operators 129

24 User-programmable keys 131

For the more advanced 131

25 BASIC keywords 133

BASIC keywords 134

26 Cassette file handling 195

27 Error handling 198

28 Merging BASIC programs 200

29 Assembly Language 202

Introduction 202
Registers in the 6502 203
Accumulator 203
Index registers X and Y 204
Program counter 204
Stack pointer 204
Flags register 205
Addressing modes 206
Entering assembly mnemonics 210
Assembly 211
Execution by USR 214
Execution by CALL 216
Quadruple precision addition 220
Multiplication 221
Division 225
Error trapping in assembler 227
Operating system calls from assembler 228
Use of operating system calls 230
OSWRCH entry: &FFEE vector: &20E 230

OSASCI entry: &FFE3 232
OSNEWL entry: &FFE7 232
OSRDCH entry: &FFEØ vector: &21Ø 233
OSCLI entry: &FFF7 vector: &2Ø8 233
OSFIND entry: &FFCE vector: &21C 234
OSBPUT entry: &FFD4 vector: &218 234
OSBGET entry: &FFD7 vector: &216 234
OSFILE entry: &FFDD vector: &212 234
OSBYTE entry: &FFF4 vector: &2ØA 235
OSWORD entry: &FFF1 vector: &2ØC 238
Events 242
Assembly Language mnemonics 243

Appendix A
VDU codes 265

Appendix B
Error messages 269

Appendix C
Operating system calls 278

Appendix D
*FX calls 28Ø

Appendix E
Fast and efficient programs 284

Appendix F
ASCII displayed character set and control codes 285

Appendix G
Text and graphics planning sheets 287

1 What is a computer?

A computer is a general purpose electronic machine that can be instructed to do a great variety of things – play games, perform complex calculations, store and retrieve information, display graphs and so on.

You can ask a computer to do things directly – by typing commands on its keyboard – but for complex tasks, a whole series of instructions is usually written and stored in the computer's memory. The computer can be instructed to call up these instructions one by one and carry them out, very fast. (Your Electron can carry out, or 'execute', over 250,000 separate instructions every second.)

A series of instructions like this is called a *program.* Programs can be recorded onto cassette by using a suitable cassette recorder in much the same way as you might record a piece of music. The main difference is that the recording is made from a computer, and is played back into the computer again. You can buy pre-recorded programs which have been written by other people, and to start you off, several programs are provided on the Introductory Cassette which comes with your Electron.

The first part of this book describes how to set up your computer, and load and run the programs on the Introductory Cassette. For information on other programs available for the Electron (the general name for programs is 'software'), write to:

Acornsoft Limited
4a Market Hill
Cambridge CB2 3NJ

The remainder of this book (chapter 4 onwards), and the book *Start Programming with the Electron,* will tell you how to write your own programs on your Electron computer. You do not need to know this in order to use your computer, as there are many commercially available programs – but we hope that you will be interested and will want to find out more about your Electron.

2 Getting started

Welcome to the Acorn Electron Microcomputer!

Chapters 2 and 3 explain how to connect your Electron to the mains and to a television or monitor. Please read them carefully before continuing.

Checklist of items

Apart from this User Guide, you should have the following items in the box you've just opened:

- An Electron Microcomputer
- A mains adapter
- Guarantee registration card
- An aerial lead about two metres long for connecting the computer to your television set
- The Introductory Cassette
- A book called *Start Programming with the Electron*

If any of these items are missing, please contact your supplier.

Additional items
You will also need the following:

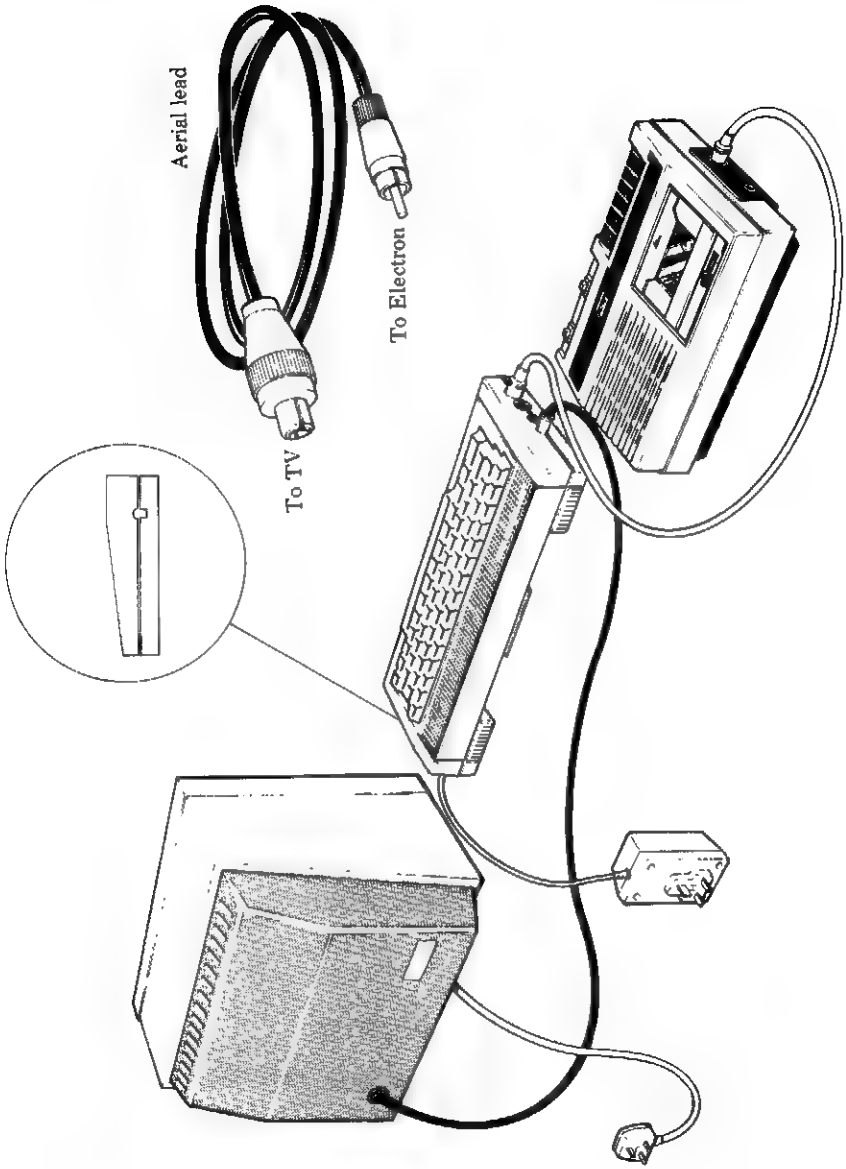
- A television set, or a good quality monochrome or colour monitor
- A mono (or stereo) cassette recorder, preferably with these facilities:

  An external motor control facility
  Record/playback socket(s) where the playback volume is controlled by the volume control

If you are going to buy a cassette recorder specifically to use with your Electron, your supplier will be able to recommend a suitable machine. However, most domestic machines can be used with good results, and chapter 3 gives details on how to connect a cassette machine to the Electron.

Connecting the Electron to your television set

Please refer to the diagram.


Aerial lead
To TV
To Electron

If you are using a monitor rather than a television, then ignore this section, and read the section opposite, 'Connecting the Electron to a monitor'; otherwise read on.

You need to use the long TV lead provided to connect the Electron to your television set, and one end of this should be plugged into the aerial socket on the back of your television set, having first removed any aerial lead already connected. If your television uses its own aerial mounted on the set, you will need to find the aerial socket marked AERIAL, or ANT, or UHF etc.

The other end of the lead should be plugged into the socket marked 'UHF TV' on the left hand side of the Electron case. If you look at the case, you will see four sockets side by side. If you then look at the bottom of the case underneath the sockets, you will see that the name of each socket is engraved there. (The 'UHF TV' socket is the one on the extreme left.)

Now switch on the television.

Connecting the Electron to the mains

The mains adapter which comes with the Electron has one lead which should be plugged into the socket on the right hand side of the Electron case. This socket is engraved '19V AC POWER IN'. The mains adapter itself can be plugged into a domestic 13A socket.

Having connected the mains adapter to the Electron and to the mains, the computer is now ON. As soon as the Electron switches on, you will hear a 'bleep', and the yellow light on the left hand side of the keyboard comes on. If this does not happen, first check that your 13A socket is working and is switched on (if it has a switch), and, if it still does not work, contact your local dealer.

Tuning the TV to the Electron

The Electron should now be connected to the mains adapter and switched on, and the TV lead should be connected to your television and to the Electron. The next thing to do is to tune your television.

First of all, turn the TV volume control to minimum - the Electron provides its own sound.

Most TVs have either a number of push-buttons or a single tuning knob for selecting TV stations.

Push-button tuning
Choose and select a push-button you don't normally use, and turn its tuning knob as far as you can in one direction, then turn it slowly the other way until the following message (or something very similar) appears on your television screen:

```
Acorn Electron

BASIC

>
```

Single tuning knob
Turn the tuning knob as far as you can in one direction, then turn it slowly in the other direction until something like the message above appears on your television screen.

Connecting the Electron to a monitor

This section only applies if you are using a monitor rather than a TV.

Monochrome monitor
You will need to acquire a special lead which should be available from the supplier of the monitor. The end of the lead which plugs into the Electron should be inserted into the socket marked 'VIDEO' on the left hand side of the Electron case. If you look at the bottom of the case, you will see that the name of each socket is engraved there.

Colour (RGB) monitor
A special lead should be supplied with the RGB monitor to plug into the Electron. The end of the lead which plugs into the Electron should be inserted into the socket marked 'RGB' on the left hand side of the Electron case. If you look at the bottom of the case, you will see that the name of the socket is engraved there.

Now try something

Take a look at the TV screen. If you have already pressed any keys on the Electron's keyboard you will probably see something unintelligible

displayed on the screen. To remove this, press the key on the keyboard marked **BREAK**. Then press any keys you like on the keyboard – as many as you like – you cannot damage the computer whatever you press! The little flashing line on the screen is called the CURSOR and is to show you where the next character you type will appear.

If you press the **RETURN** key at any stage, a new line is started and the computer will probably display a message on the screen immediately underneath what you've typed, such as

```
Mistake
```
or
```
Syntax error
```

or something else. Don't take any notice of this; the Electron's command of the English language isn't quite as good as yours! Later on we will show you how to tell the Electron to do things for you by using its own language called BASIC.

If you press a key and hold your finger on it, you will notice that after a short time the character displayed on the screen repeats itself over and over until you take your finger off the key again.

If the yellow light to the left of the **CAPS LK** key is still on, then letters of the alphabet will appear on the screen as capital ('upper case') letters.

To type small letters, press the **SHIFT** key and hold it down while you press the **CAPS LK** key. The yellow light goes out, and you can now type 'lower case' letters on the screen.

To get back to capitals, press **SHIFT** and **CAPS LK** again – the keyboard is now locked into producing 'upper case' letters again.

Spend some time playing with the keyboard if you aren't very familiar with the layout, and if anything strange happens, just press the **BREAK** key. This will clear the screen and you can continue. When you get to chapter 5 the keyboard operation is explained in detail.

The different typefaces used in this book represent the following:

- Ordinary text appears like this, or like *this* for emphasis.
- Text typed into the computer or displayed on the screen appears `like this` or **like this**.
- Words like **RETURN** mean that you press the key marked RETURN rather than actually type the letters R E T U R N .

3 Using a cassette recorder

Introduction

When you have learnt to write programs that enable the Electron to do things for you, it will soon become obvious that you need to make a copy of these programs for future use. One very good reason being that when you switch off the Electron, everything you typed into the computer is forgotten. The Electron gives you the facility of recording your programs onto cassettes for future use (just as you might record music).

Not only this, but you can then play back pre-recorded programs for the Electron which other people have written, and make them work on the Electron.

This chapter tells you how to set up your cassette recorder for saving or loading programs. These programs will show you some of the things the Electron Microcomputer can do.

First of all, you need to connect a suitable cassette recorder to the Electron.

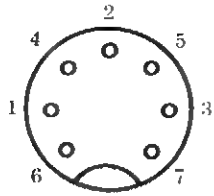
Connecting a cassette recorder

The sort of lead you need to connect the Electron to a cassette recorder depends on what type of sockets are fitted to your cassette recorder. One end of the lead must have a 5-pin or 7-pin DIN plug fitted, and this plugs into the socket on the left hand side of the Electron case marked 'CASSETTE' – you will find the name engraved on the bottom of the case immediately below the socket.

The wiring diagram overleaf shows the wiring of the CASSETTE socket on the Electron as you look at it from the mating side.

Your nearest Electron dealer or hi-fi dealer should be able to help provide the correct lead if you have any problems, especially if you show him the drawing.

Cassette interface



7-pin DIN socket viewed from outside of case

PIN
1 OUTPUT TO CASSETTE (RECORD) LINKED TO PIN 4
2 COMMON
3 INPUT FROM CASSETTE (PLAY)
4 OUTPUT TO CASSETTE (RECORD) LINKED TO PIN 1
5 NOT CONNECTED
6 MOTOR CONTROL SWITCH
7 MOTOR CONTROL SWITCH

To help you out of immediate difficulties however: if your cassette recorder has a 5-pin DIN socket for record/playback, then you can use a standard 5-pin DIN to 5-pin DIN lead between the Electron and the cassette recorder. The only disadvantage of this is that the Electron will not be able to control the cassette recorder's motor (assuming that your cassette recorder has the facility for externally pausing the tape while it's playing). This facility isn't absolutely necessary, but it does mean that instead of the Electron stopping and starting the tape automatically, you will have to do it manually.

If you find that you are listening to the Introductory Cassette via the cassette machine's internal speaker, you may want to insert a plug into the earphone or external loudspeaker socket to stop it – computer programs sound like a screeching noise which is not pleasant! If in doubt, ask your local hi-fi dealer how to do this on your particular machine.

Motor control

If your cassette recorder has a motor control facility and you are using it, then you can ignore any messages asking you to stop the tape – this will be handled by the Electron automatically. All you need to do is to press the PLAY button on the cassette recorder when you first start loading the first program (explained in chapter 4) and leave it on.

If your cassette recorder does not have motor control, then you must stop the tape as indicated by the computer.

You are now ready to use the Introductory Cassette.

4 The Introductory Cassette

The Introductory Cassette contains lots of interesting demonstration programs which are recorded on both sides of the cassette. If you start at the beginning of side A and follow the directions in this chapter, the computer will take you through each of these programs in turn. When you get to the last program on side A, you will be asked to turn the cassette over and continue on side B.

You will notice that the programs on side B start about a third of the way along the tape. The reason for this is that there are four extra programs at the beginning of side B and at the very end of side A which relate to the book *Start Programming with the Electron*. When you have got more familiar with your Electron and start using this book, then rewind the tape to the beginning of side B and follow the directions given in the book.

This chapter deals with the demonstration programs only, so insert the cassette into your cassette recorder – side A uppermost.

Adjusting the volume control and loading the first program

On some cassette recorders, the volume control setting must be adjusted first before the Electron can 'hear' the programs being played. If this is the case for your machine, set the cassette recorder volume control to about two-thirds maximum, and the tone control (if fitted) to maximum. See if the yellow light on the left of the keyboard is on, and if it isn't press the **SHIFT** key and the **CAPS LK** key down together – this will make the yellow light come on. Press **BREAK** to ensure that the computer is completely reset. Now type the following *exactly* as it is printed below.

```
CHAIN "INTRO"
```

To type each quotation mark, hold your finger on the **SHIFT** key and press the key with the number 2 on it (immediately above the 2 is the " character). Make sure you type an 'O' and not a zero.

Then press **RETURN**. If you make a mess of it, don't worry – just press **ESCAPE** and try again.

Press PLAY on the cassette recorder, and the tape should start moving. The message

```
Searching
```

should appear on the screen. This means that the computer is looking for the program called "INTRO" on the cassette. As soon as it has found it, you should see this message displayed on the screen:

```
Loading
INTRO      00
```

This means that the program called INTRO has been found, and the computer is loading it (copying it) into its memory. The program is recorded in 'blocks' on the cassette, and the numbers on the screen next to **INTRO** tell you which block is being loaded at the moment.

If the message above doesn't appear on the screen after about 30 seconds from pressing PLAY, then turn the volume control up a bit more, and wait for about ten seconds. If there's still no message, turn it up more and keep on until you get a message similar to the one above.

The two numbers to the right of **INTRO** will be higher than **00** by this time, so completely rewind the tape and start again. Press **BREAK** and retype

```
CHAIN "INTRO" RETURN
```

Now that you have found the right setting for the volume control, there is no need to adjust it again.

Once INTRO has been loaded successfully, four numbers appear to the right of the two numbers already there. When this happens, it means that the program has finished loading, so unless the Electron has done it for you already, stop the tape. If you don't, then you'll have to rewind the tape back to the end of the INTRO program before the Electron can load the next program. If for any reason the program did not load successfully, a message will appear telling you to rewind the tape and start again.

After a short pause, the INTRO program starts running.

The INTRO program uses some of the colour graphics and sound capabilities of the Electron, and also includes an index of the programs you are about to see on side A of the Introductory Cassette.

Here is a quick guide to the other demonstration programs:

KEYBOARD
This program will help you to get to know the Electron's keyboard. You will be asked to type different characters from the keyboard. So that you can judge your performance and see how you improve, the computer times you!

SKETCH
Feeling creative? Here's a chance to put your artistic talents to use. In the centre of the screen there is a cross which you can move wherever you like. As the cross moves, it draws a line in the colour of your choice. You can also move the cross without drawing a line – rather like lifting your pen off the paper – and then carry on. The keys you can use and what they do are listed at the bottom of the screen.

PIANO
The Electron turns itself into a musical instrument. At the bottom of the screen there's a picture of a piano keyboard with the corresponding keys on the Electron keyboard shown. At the top of the screen, the musical score appears as you play.

DODGEMS
You are in control of a car driving through a maze of roads which each contain a row of dots. You must drive along every road and clear the dots on them to score the maximum number of points. Unfortunately, there's a computer car coming the other way whose sole purpose in life is to crash into you!

You control the car with five keys which are described at the beginning of the game, and these allow your car to:

– Go left
– Go right
– Go up
– Go down
– Go faster

If you are going too fast, you have to turn at every junction unless you slow down in time.

You score one point per dot and one more when they're all gone. Once there are no dots left, a new maze appears and you carry on as before –

only this time the computer car travels faster!

BIORHYTHMS

This ingenious program plots your 'biorhythms' which is the supposed balance between your emotional, physical and intellectual states. Some people believe that these are regular cycles which show when you are at your best and worst physically, emotionally and intellectually. They also believe that the rhythms started at birth and can be predicted mathematically.

All you need to do is enter your date of birth and the date you would like the biorhythm chart for – perhaps today, or maybe you have an important event coming soon, and you want to find out how you'll feel on that day.

The program calculates these cycles from your birth and then displays a chart which indicates your state of well-being on the chosen day. Biorhythms or no biorhythms, this program demonstrates the computer's calculation speed (for example, the number of days from your birth) and how the computer can be used to display graphical information.

CLOCK

This program shows that the computer has more of a memory than you think. Remember the computer asked you the time in the INTRO program? If you typed in the correct time then, you can check it now – either as a digital or analogue read-out. You can even reset it if you want. As well as demonstrating the Electron's high resolution graphics, this program also shows that the computer is an excellent time-keeper.

GOMOKU

Gomoku is a very old board game where two opponents (you and the Electron) try to produce a row, column or diagonal with five counters. It is really a sophisticated version of noughts and crosses where you must plan your moves carefully – there is a very large number of possible moves.

The game starts with a blank screen and the message

```
SHALL I START?
Press Y or N
```

After you've pressed Y or N, the board appears on the screen, and if you let the computer have first go it will have placed its counter somewhere

on the board.

A small cross shows where your counter will be placed. Once you are happy with the position for your counter, press **RETURN**, and a counter appears in place of the cross.

The computer has another go and play continues until one of you manages to get an unbroken line of five counters on the board. The winning line flashes for a few seconds, and if you want another game, press the space bar.

MESSAGE

This program tells you to stop the tape (if you haven't got motor control), and continue on side B of the cassette where you will find the programs described below.

PATTERNS

This program generates kaleidoscopic patterns in colour, and no two patterns are ever quite the same. To start a new pattern, press the keys marked 1 and 2 in sequence. The pattern itself and its colours are randomly selected each time the keys are pressed and serve to demonstrate the Electron's high resolution colour graphics. If you want to sit back and watch, press one of the keys for a second or two; the computer will continue to generate the patterns until the repeat action of the key you pressed runs out.

MARSLANDER

You are in command of a spacecraft which you must try to land on a flat section of the Martian terrain as gently as possible. Key X rotates the craft clockwise, and key Y anticlockwise. The space bar fires the rocket motor and makes the spacecraft move in whatever direction it is pointing. To land the spacecraft safely, it must be pointing upwards, at a speed of less than 5Ømm/s and you must touch down on a flat section of Mars. The score depends on your speed when you land, with a possible 5ØØØ points for landing at Ømm/s. After a successful landing, the computer will tell you what sort of landing you made, your touch-down speed, how much fuel you have left, and ask you to take off for another landing site which is more than a specified distance away. On each successful landing, you get 3Ø extra fuel units and your old score is added to your new one.

BUGZAP
This program gives you some target practice shooting down a space invader. The keys on the keyboard marked Z and X move the launcher, and the space bar fires missiles at the space invader. Good luck!

ISLAND
This program draws a colour picture of a desert island with palm trees, and a moving sea which occasionally becomes turbulent. This program demonstrates the use of animation in graphics.

PLANETS
This is a very striking example of high resolution colour graphics on the Electron, where animation is used to produce a marked three-dimensional effect.

Loading each program from the cassette

If you simply follow the instructions on the screen, you will be loading each program into the Electron's memory one after the other. If you want to locate a particular program on the cassette, wind the tape to any point before the program you want (a tape counter can be of great help here) and type

`CHAIN "XXXX"` (where XXXX represents the program name)

Then press the **RETURN** key. The computer will look through the tape until the program name is found, then it will start loading the program, and afterwards, run it.

Alternatively, position the tape just before the beginning of the program you want, and type

```
CHAIN ""
```

Then press the **RETURN** key. The computer will load and run the next program it finds on the tape. To help you find the beginning of a program, listen to the tape; each program sounds like a screeching noise, and between programs you will hear a continuous high-pitched tone. If you start loading during the high-pitched tone then you know you're in the right place.

There are two important points to bear in mind. The first is that the word **CHAIN** is a command which tells the computer to load the program you specify and then to execute it (run it). So in fact the **CHAIN** command tells the computer to do two things. The second point is that when the computer is searching for the program you specified, it will still display on the screen the name of whatever program is currently playing. This should give you an idea of how close you are to the program you want. However, remember that whatever program is displayed on the screen, the computer isn't loading it *unless* you see the word

```
Loading
```

above the program name. Don't be fooled into thinking that just because a program name appears on the screen, it is being loaded (copied) into the Electron's memory.

Chapters 9 and 26 give a detailed account of all the facilities made available to you by the Electron when recording (saving) and playing back (loading) programs to and from cassette.

5 How to use the keyboard

Introduction

The Electron keyboard works rather like an ordinary typewriter keyboard. The main difference is that when you press a key, instead of letters appearing on a sheet of paper, they appear on your television screen.

The small white flashing line you can see on your TV screen is called the CURSOR and it shows where the next character to be typed will appear on the screen.

One thing to bear in mind which the keyboard doesn't make obvious – the keys which produce the letters of the alphabet

Q W E R T Y U I O P
A S D F G H J K L
Z X C V B N M

can be made to produce small letters as well as capital letters.

As you can see, virtually all the keys have more than one character/word printed on them – some even have three! There are four keys on the keyboard whose purpose is to sort out which one appears on the screen when you press a particular key. These are

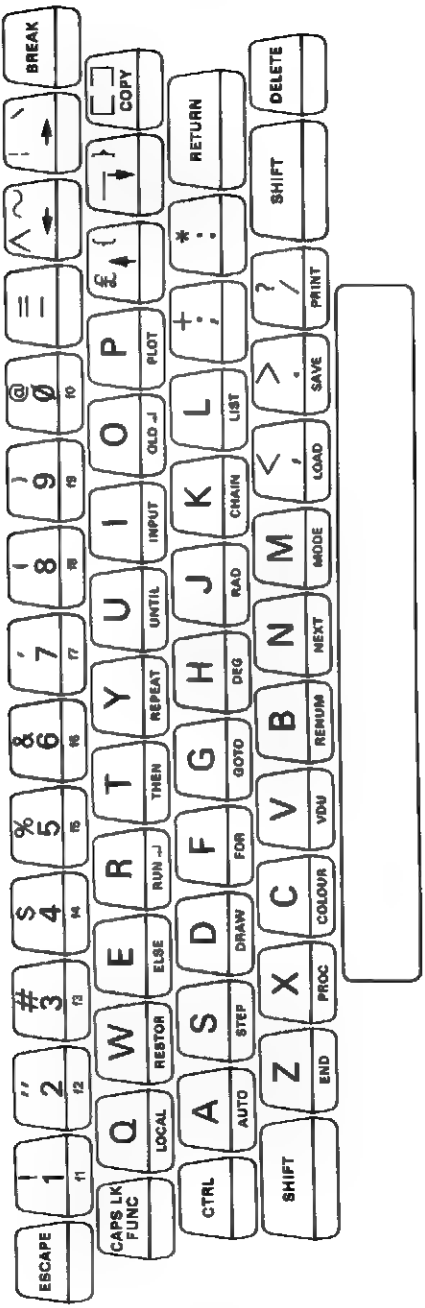
SHIFT – there are two of these keys

CTRL

CAPS LK / **FUNC** – this is one key, which is normally 'FUNC' but is 'CAPS LK' when used with the SHIFT key.

These keys are used in various combinations to give you the character or word you want to appear on the screen, and this is described shortly.

The **BREAK** key clears the screen and prepares the computer for entering a new program. Any program lines already entered will be deleted (but can be recovered by using the **OLD** command – see chapter 25).


ESCAPE
CAPS LK FUNC
CTRL
SHIFT
SHIFT
RETURN
DELETE
COPY
BREAK
LOCAL
AUTO
STEP
PROC
END
RESTOR
ELSE
DRAW
COLOUR
RUN ↵
THEN
FOR
GOTO
VDU
REPEAT
DEG
RENUM
UNTIL
RAD
NEXT
INPUT
CHAIN
MODE
OLD ↵
LIST
LOAD
PLOT
SAVE
PRINT

Holding down the **CTRL** key then pressing the **BREAK** key clears the screen and restores the computer to how it was when you first switched it on.

The **ESCAPE** key is used to interrupt a sequence of events generated by the Electron, such as a program listing or program execution etc.

The **DELETE** key moves the cursor backwards along a line and deletes whatever character it finds there.

Choosing the keyboard characters

This section tells you how to select all the different characters on the keyboard.

SHIFT and CAPS LK

Press the **SHIFT** key, and hold it down while pressing the **CAPS LK** / key. If you keep on doing this, you will notice that the yellow light to the left of the keyboard goes on and off.

When the light is on, the keyboard is in 'CAPS LOCK', and this means that pressing letter keys will give you capital letters, but all other keys will give you the lower of the two black characters on the top of the key.

When the light is off, the keyboard is no longer in 'CAPS LOCK' – the only difference is that the letter keys will give you small letters when pressed.

SHIFT

With the yellow light off, press the **SHIFT** key, and hold it down while you press one of the character keys. All letters appear on the screen as capitals, and all the keys with two black characters marked on them make the upper character appear on the screen. The keys with three black characters marked on them make the top left character appear on the screen.

With the yellow light on, press the **SHIFT** key, and hold it down while you press one of the character keys. Even though the keyboard is in 'CAPS LOCK', all letters appear on the screen as small letters.

FUNC

A lot of keys have light brown characters marked on the front, such as GOTO, RUN, f7 etc. Press the / **FUNC** key and hold it down: at the same time, press the key with PRINT marked on the front. The word **PRINT** appears on the screen. Try pressing the / **FUNC** key with some of the other keys with light brown writing on them.

You probably noticed that the keys marked 'f0' to 'f9' won't do anything when you press them. This is because they are 'user definable' keys which means that you can choose what they will do when they are pressed. How you define them is described in chapter 24.

If you have never come across the computer language called 'BASIC', you may wonder what the point is of being able immediately to display these words on your TV screen. The answer is simple; the Electron computer understands and uses the 'BASIC' language, and these words are the most common 'keywords' (or commands) in this language, so rather than having to keep on typing these out in full, you can simply press the / **FUNC** key and the relevant character key. This saves a lot of time when typing lengthy programs into the computer.

The arrow keys and the COPY key

There are five keys in the top right area of the keyboard which have three black symbols marked on them (four of the keys have arrows on them, and one of them the word **COPY**). First of all, here is how to select which symbol you want from any of these keys:

For example, the key marked



Key by itself	selects left arrow (which moves the cursor)
SHIFT and key	selects ∧ and produces it on the screen
CTRL and key	selects ~ and produces it on the screen

What the arrow and COPY functions do

If you press any of the keys marked with arrows, the cursor moves in the

direction of the arrow, and leaves a square block where it was before it moved.

If you now press the **COPY** key, both the 'line' cursor and the 'block' cursor move across the screen, and every character above the 'line' cursor is copied to where the 'block' cursor is. This is a very useful facility for editing programs: the idea is to copy the line with a mistake in it onto a fresh line, and correct the mistake(s) while you're doing it. Editing programs using this facility is described in chapter 7.

Summary

1. CAPS LOCK off (yellow light off)
   Small letters and bottom black characters

2. CAPS LOCK on (yellow light on)
   Capital letters, otherwise as above

3. CAPS LOCK on (yellow light on) **SHIFT** with **key**
   Small letters and top (left) black characters

4. CAPS LOCK on or off **FUNC** with **key**
   BASIC keywords, as indicated on the front of the keys, are produced

5. CAPS LOCK on or off **CTRL** with **key**
   Selects top right hand character from keys with three black characters marked on them.

Note: See Appendix A for a list of VDU control codes which can be generated by pressing **CTRL** and some of the other keys on the keyboard.

6 Introducing commands and programs

The Electron computer, like any computer, has to be told what to do before it will do anything for you. The only way you can 'talk' to it is by typing *commands* to it on the keyboard, and the Electron tells you what you typed by displaying it on the TV screen.

The Electron understands two languages, one called BASIC and the other called *Assembly Language*. As these are written languages only – in other words they are meant to be typed into the computer rather than spoken to it – they each have their own special vocabulary and grammar.

Talking to the computer in Assembly Language is no easy task for the beginner so it is discussed in a separate section towards the end of this book. From now on, we will be helping you to speak to your computer in the language called BASIC. This language consists of a number of words or commands. Some of these are printed on your Electron keyboard in light brown letters.

Note that the Electron displays a > sign followed by the flashing cursor. This is called a 'prompt', and means that the computer is waiting for your instructions. Normally the Electron will prompt you when it is waiting for you to type something in. The > prompt means that the Electron is expecting a BASIC command.

Press the **BREAK** key to clear the screen, and type the following line

```
PRINT "HELLO"
```

Then press **RETURN**. As soon as you press the **RETURN** key, the computer obeys the BASIC command **PRINT**, and because the rest of the line was in quotation marks it displayed the word **HELLO** on the screen.

Now type this

```
PRINT "I'M"
PRINT "LEARNING"
PRINT "BASIC"
```

DON'T FORGET TO PRESS THE **RETURN** KEY AT THE END OF EACH LINE!

As you can see, the Electron reads each command and executes it immediately after you press the **RETURN** key. You can also type in all these commands first AND THEN make the computer execute them when you tell it to do so. This is done by putting a number in front of each instruction.

Now type the following

```
10 PRINT "HELLO"
20 PRINT "I'M"
30 PRINT "LEARNING"
40 PRINT "BASIC"
```

DON'T FORGET TO PRESS THE **RETURN** KEY AT THE END OF EACH LINE!

The computer hasn't carried out your instructions like it did before, so now type

```
RUN
```

Followed by the **RETURN** key.

This time the computer has printed your message on the screen all in one go.

Congratulations, you have just run your first program on the Electron Microcomputer!

So a program is simply a collection of numbered instructions. The line numbering has two purposes: firstly to tell the computer not to execute each line after you have pressed the **RETURN** key, and secondly to help the computer decide in what order it should execute the instructions – after you have typed **RUN** of course.

The actual numbers you type in can be any numbers you like as long as you remember that the computer will execute the program lines in numerical order.

The chapters which now follow serve as an introduction to the Electron BASIC language, and how to use the facilities it offers you. The chapter

on editing your programs will help you speed up the process of typing in programs and making changes to get them working.

Unlike a typewriter, you *don't* need to press the **RETURN** key when what you have typed has filled up the current line on the screen – just carry on typing. What happens is that the computer starts a new line on the screen, and the subsequent characters you type are displayed on the new line. Prove this to yourself by typing a lot of characters.

Remember that pressing the **RETURN** key tells the computer that you have reached the end of the command or program line you have just typed. If a command, the computer then executes it, and if a program line, the computer stores it in its memory.

What is hexadecimal?

Once you get more familiar with the Electron, you may come across things called 'hexadecimal' numbers. Here is a brief explanation of what they are.

Hexadecimal numbers, sometimes called 'hex' numbers for short, have sixteen separate digits, compared to our decimal numbers which only have ten (including the number zero). This is how you count in hexadecimal, with the decimal equivalent underneath.

```
Ø  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F  1Ø 11 12...
Ø  1  2  3  4  5  6  7  8  9  1Ø 11 12 13 14 15 16 17 18...
```

To show you how to carry on counting in hexadecimal:

```
12  13  14...19  1A  1B  1C  1D  1E  1F  2Ø...2F  3Ø...
3F  4Ø.....9Ø...9F  AØ  A1  A2...AF  BØ...BF  CØ.....
FF  1ØØ  1Ø1...etc
```

To help you and the Electron tell the difference between decimal numbers and hexadecimal numbers, you should always type an '&' sign in front of a hexadecimal number. If you don't, then the computer will assume your number to be a decimal. So we can now say &AØ = 16Ø (hexadecimal AØ is equal to decimal 16Ø).

7 Editing programs

Introduction

The Electron provides you with a number of very useful facilities for laying out, editing and listing your programs. If you haven't done any programming before, here is a brief list of the sort of facilities you will need when typing in programs and making them work:

- Being able to display part or the whole of your program on the screen whenever you want to.
- Correcting mistakes, or editing.
- Putting comments or notes into the program to help you remember what each part of the program is doing.
- Deleting one or more program lines.

To start looking at these facilities and how to use them, type in the sample program below which we will use to demonstrate the different facilities.

First, press **BREAK** to clear the screen and reset the computer, then type the following, and take care with the punctuation and spaces in the last line.

```
10 PRINT "GIVE ME A NUMBER BETWEEN ONE AND
TEN"
20 INPUT X
30 Y=2*X
40 PRINT "TWO TIMES ";X;" IS ";Y
```

After typing in the above program, type

```
RUN RETURN
```

When you run this program, the following happens

line 10 **GIVE ME A NUMBER BETWEEN ONE AND TEN** appears on the screen

line 20 A question mark appears on the line below, and the computer waits for you to type in a number which is stored as a variable called X. Type in a number and press **RETURN**

line 3Ø The computer multiplies X by 2 and stores the result as a variable called Y

line 4Ø The following is printed on the screen: **TWO TIMES** (the number you typed in) **IS** (the result)

If the program won't work properly, or you get an error message, press **ESCAPE** and type it in again – you most likely made a mistake when you typed it in the first time.

Listing the program

When you want to change your program in any way, you will need to display the program (or at least the bit you want) on the screen. To do this, use the BASIC command **LIST**. Type

```
LIST RETURN
```

Your program appears immediately underneath the **LIST** command on the screen.

If you only want to look at one particular line, say line 4Ø, type

```
LIST 40 RETURN
```

Line 4Ø of your program is displayed on the screen.

To look at a number of consecutive lines, say lines 2Ø to 4Ø, type

```
LIST 20,40 RETURN
```

Lines 2Ø, 3Ø and 4Ø appear on the screen.

If you want to see from the beginning of the program up to a particular line, say line 3Ø, type

```
LIST ,30 RETURN
```

Lines 1Ø, 2Ø and 3Ø appear on the screen.

If you want to see from a particular line to the end of the program, then type

```
LIST 20, RETURN
```

Lines 2Ø, 3Ø and 4Ø appear on the screen.

Please refer to chapter 25 for a description of the **LISTØ** commands. These commands provide you with even more facilities when listing programs.

Editing programs

There are three ways of correcting mistakes in programs you have typed.

One of these you have already met in chapter 5: that is, pressing the **DELETE** key which moves the cursor back along the current line deleting each character as it goes. There is one major drawback to this method – if you have finished typing a line and have pressed **RETURN**, you can't get the cursor to go back to that line by just pressing the **DELETE** key. As we said before, pressing the **DELETE** key only moves the cursor back along the *current* line, which may not be the one you want to correct.

Another method is to type in the line again, but with the correction. The computer always replaces the old program line with any new version you type in. If the line to be corrected is very short, then this method is fine; but if the line is long or complicated, then use the third method described below.

Editing with the arrow keys and the COPY key

Type

```
LIST RETURN
```

The program appears on the screen, and we are going to use it to try out some editing. The following should now be on your screen:

```
>LIST
    10 PRINT "GIVE ME A NUMBER BETWEEN ONE A
ND TEN"
    20 INPUT X
    30 Y=2*X
    40 PRINT "TWO TIMES ";X;" IS ";Y
>_
```

Supposing you want to change the word **BETWEEN** to **FROM** in line 1Ø.

First of all, press the up-arrow key five times. The original cursor position under line 4Ø becomes a white square, and the cursor moves up to line 1Ø.

Now press the key marked **COPY** three or four times. The cursor moves along line 1Ø, the white square moves along as well – and line 1Ø is copied underneath line 4Ø. Keep on pressing the **COPY** key until you have copied the word **BETWEEN**, then stop. Note that if you hold the key down, the repeat action allows you to move the cursor quickly across the screen. A quick press and release gives you precise control, moving just one character position. The following should be on your screen:

```
>LIST
   1Ø PRINT "GIVE ME A NUMBER BETWEEN_ONE A
ND TEN"
   2Ø INPUT X
   3Ø Y=2*X
   4Ø PRINT "TWO TIMES ";X;" IS ";Y
>  1Ø PRINT "GIVE ME A NUMBER BETWEEN■
```

Now press **DELETE** until **BETWEEN** has been deleted from the new line 1Ø.

NOTE THAT THE CURSOR ON THE OLD LINE 1Ø HASN'T MOVED.

If the cursor isn't in the right place, ie underneath the space separating **BETWEEN** and **ONE**, move it there now by using the arrow keys.

Now type in the word **FROM**, then press the **COPY** key to copy the rest of line 1Ø to your new version.

Press **RETURN**. The white square disappears, and the cursor goes to the start of a new line. The result should be this:

```
>LIST
   1Ø PRINT "GIVE ME A NUMBER BETWEEN ONE A
ND TEN"
```

```
   20 INPUT X
   30 Y=2*X
   40 PRINT "TWO TIMES ";X;" IS ";Y
   10 PRINT "GIVE ME A NUMBER FROM ONE AND
TEN"
>_
```

If you **LIST** the program again, by typing

LIST **RETURN**

you will see that the new line 10 has replaced the old version. Have a go at editing line 10 again and change **AND** to **TO**.

There are no restrictions on how much you move the cursor around when you're copying. This means that you can copy bits from lots of different lines onto your new line all in one go: wherever you move the cursor to by using the arrow keys, you can then copy as much as you like of that line, then move the cursor somewhere else and continue copying.

Deleting lines from your program

There are two ways of deleting whole lines from your program.

The first method is to type in the line number of the line you want to delete, then press **RETURN**. What you are doing is entering a new version of that line into the computer – only with nothing in it. Because the computer always replaces a previous version of a line with any new version you type in, this method effectively deletes that line number.

For example, to delete line 10 of the program, type

10 **RETURN**

Now list the program by typing

LIST **RETURN**

And you will see that line 10 has been deleted. Now type the original line 10 again so that it goes back into your program.

The second method of deleting program lines is to use the **DELETE** command. This command allows you to delete a number of consecutive lines.

To do this, type **DELETE**, then type the first line number to be deleted, then a comma, followed by the last line to be deleted. The following examples will help you understand, but before typing them into the computer, remember that they will in fact delete parts of your program, so afterwards you will need to type the deleted lines again to restore your program!

First of all, list your program so that you can see it on the screen. You should have lines 1Ø, 2Ø, 3Ø and 4Ø. If not, then replace the missing ones (copy them from the program listing at the beginning of this chapter).

To delete lines 1Ø, 2Ø and 3Ø, type

```
DELETE 10,30 RETURN
```

To delete all lines from the beginning of the program to a particular line number, type Ø followed by a comma followed by the last line to be deleted. For example, to delete lines 1Ø, 2Ø and 3Ø, type

```
DELETE Ø,30 RETURN
```

To delete all lines from a particular line to the end of the program, the numbers to enter after the **DELETE** command should be the first line number to be deleted, then a comma and then any number which you know to be equal to or greater than the last line number of the program. For example, to delete line 2Ø onwards, type

```
DELETE 20,100 RETURN
```

Inserting new lines into your program

First of all, list your program (if you haven't deleted it all!) and type in any lines which may be missing.

Having typed in the first attempt of a program, executed it by using the **RUN** command, and changed or deleted lines as necessary to make the program work, you may want to insert new lines. You will then see how important it was to leave plenty of unused line numbers between the original lines in your program. To insert new lines, decide when you want

the new line to be executed by the computer when it runs the program, then choose a suitable line number. For example, to insert a line which makes the computer print another message underneath **GIVE ME A NUMBER FROM ONE TO TEN**, you will need to insert a new line somewhere between lines 10 and 20. Let's choose 15; this still leaves some room either side for any more new lines, so now type

```
15 PRINT "NICE DAY ISN'T IT" RETURN
```

Now list the program, and you will see that the new line appears in the listing. The listing should look like this

```
10 PRINT "GIVE ME A NUMBER FROM ONE TO TEN"
15 PRINT "NICE DAY ISN'T IT"
20 INPUT X
30 Y=2*X
40 PRINT "TWO TIMES ";X;" IS ";Y
```

Make the computer execute the program by typing

```
RUN RETURN
```

and type a number for it to multiply by two. Remember that the computer is expecting a numeral and will not recognise letters. This is because our little program would need some more lines adding to it before the computer would recognise THREE instead of 3, or TEN instead of 10.

Renumbering the program

There may be occasions when you want to change the line numbers of a program but without changing the order in which they are executed by the computer. The command which does this is **RENUMBER**. This facility is especially useful when you want to insert say 25 new program lines between lines 10 and 20 in your existing program.

You can specify two numbers after typing the **RENUMBER** command. The first number tells the computer what you want the first program line number to be changed to, and the second number tells the computer how much to add to each line number to get the next one.

For example

```
RENUMBER 100,20 RETURN
```

will renumber the first line as line 100, and the remaining lines will be numbered 120, 140, 160, and so on.

If you leave out the second number in the **RENUMBER** command, the computer will automatically change the second line number to ten more than the first, and then carry on through the listing. So if you had a program with line numbers

```
23 PRINT "The Electron"
24 PRINT "Microcomputer"
26 PRINT "will do"
30 PRINT "many things for you"
```

and type

```
RENUMBER 100 RETURN
```

listing the program will give

```
100 PRINT "The Electron"
110 PRINT "Microcomputer"
120 PRINT "will do"
130 PRINT "many things for you"
```

If you simply type

```
RENUMBER RETURN
```

then your program lines will be renumbered 10, 20, 30, 40, 50 and so on.

Getting the computer to number each program line

Instead of typing line numbers at the beginning of each new program line, you can get the computer to do this for you by using the command **AUTO**.

If you type

```
AUTO RETURN
```

you will see that the computer will print the number 10 on the line below. You can then type the first program line, press **RETURN** at the end, and

the number 2Ø appears on the next line – and so on. When you want to switch off this automatic line numbering, press **ESCAPE**.

If you don't want to start the program at line 1Ø, and want a different number of spare lines between each of your program lines, then you can type in two numbers – separated by a comma – after the **AUTO** command. For example, if you type

```
AUTO 4ØØ,15 RETURN
```

the program line numbers will come out as 4ØØ, 415, 43Ø, 445 and so on.

Now press **BREAK**, and retype the example program we were using previously using the **AUTO** facility. Start the program numbering at 2ØØ and continue in steps of 5Ø.

Putting notes into your programs

When typing programs, especially long ones, it is a good idea to insert comments here and there to tell you what each part of the program is doing. This is done by using the **REM** command. All **REM** does is tell the computer to ignore the rest of the line when it executes the program, but your comments will still appear in the program listing when you give the command **LIST**.

For example, we could insert a comment at the beginning of our example program to tell us what the purpose of the program is, like 'This program doubles numbers'. To insert this into the program listing, first type

```
LIST RETURN
```

to get a listing of the program. Then choose a line number less than the first program line, say 5. Now type

```
5 REM This program doubles numbers RETURN
```

If you list the program, then run it, you will see that the comment in line 5 appears in the listing, but is ignored when the program is executed.

Retrieving a program and starting a new one

If you press the **BREAK** key for any reason, the program you have typed in so far gets 'lost'. To get it back again, type

```
OLD RETURN
```

If you want to start a new program, make sure that any program already stored in the computer has been deleted. In other words, type

LIST **RETURN**

and if a program appears, then delete it by typing

NEW **RETURN**

You can now enter your new program, and the old one is forgotten.

Listing long programs

When listing very large programs, which won't fit on the screen all in one go, the beginning of the listing will disappear off the top of the screen. There are two ways of getting round this: one way is to press **CTRL** and **SHIFT** together as soon as you have typed

LIST **RETURN**

The effect of this is to halt the displayed listing on the screen. Taking your finger off either **CTRL** or **SHIFT** allows the listing to continue, and this enables you to 'step' through chunks of the listing.

The other method is to put the Electron into 'paged mode'. Press **CTRL** N to get into 'paged mode', then list the program. The listing stops as soon as it has filled the whole screen. To display the next 'screenful' of listing, press the **SHIFT** key.

Press **CTRL** O to get out of paged mode.

8 Trying out some programs

Introduction

Having seen something of what the Electron can do, and having got used to typing on the keyboard, it's now your turn to tell the Electron to do things. Because the computer does exactly what it is told, remember to type in the examples given *exactly* as they appear in this chapter. You'll find that you can sometimes get away with adding an extra space here and there or leaving one out, but rather wait until you are more familiar with the Electron BASIC language before experimenting! After you have typed in each program, type

RUN **RETURN**

to execute it. If the program doesn't appear to work, the computer may help you by displaying an error message, and telling you in which line the mistake appears. Press **ESCAPE** and list the program as described in the chapter on editing. Make the necessary alterations, then run the program again.

Before you start, press the key marked **BREAK** on the keyboard. This will get the computer ready for you, and also start you off with an empty screen.

PERSIAN

This program produces a pattern by drawing hundreds of lines. Random colours are selected by lines 40 and 50. Line 60 moves the origin (middle) of the picture to the centre of the screen. The program stops after a while, so run it again to repeat the patterns.

```
10 REM PERSIAN
20 MODE 1
30 D%=4
40 VDU 19,2,RND(3)+1,0,0,0,
50 VDU 19,3,RND(3)+4,0,0,0
60 VDU 29,640;512;
70 J1%=0
80 FOR K%=500 TO 380 STEP -40
90 REPEAT J2%=RND(3): UNTIL J2%<> J1%
```

```
100 J1%=J2%
110 GCOL 3,J1%
120 FOR I%=-K% TO K% STEP D%
130 MOVE K%,I%
140 DRAW -K%,-I%
150 MOVE I%,-K%
160 DRAW -I%,K%
170 NEXT
180 NEXT
```

POLYGON

This program draws polygons (many sided shapes) in random colours.
Lines 90 to 150 select a random place on the screen which will be the centre (origin) of the next shape.
Lines 170 to 250 calculate the X and Y coordinates of each corner of the polygon and store the values in two 'arrays' for future use.
Lines 220 and 160 fill the shape with black triangles which make it appear as if the new polygon is in front of the older ones.
Lines 260 to 310 draw all the lines that make up the polygon.
Lines 30 to 50 set the actual colours of logical colours 1, 2 and 3 to red, blue and yellow. You can change these to use other colours if you like.
Unlike the PERSIAN program, this one carries on and on until you stop it yourself. To do this, either press **ESCAPE** or **BREAK**.

```
 10 REM POLYGON
 20 MODE 1
 30 VDU 19,1,1,0,0,0
 40 VDU 19,2,4,0,0,0
 50 VDU 19,3,3,0,0,0
 60 DIM X(10)
 70 DIM Y(10)
 80 FOR C=1 TO 2500
 90 xorigin=RND(1200)
100 yorigin=RND(1000)
110 VDU 29,xorigin;yorigin;
120 radius=RND(300)+50
130 sides=RND(8)+2
140 MOVE radius,0
150 MOVE 10,10
160 GCOL 0,0
170 FOR SIDE=1 TO sides
```

```
180 angle=(SIDE-1)*2*PI/sides
190 X(SIDE)=radius*COS(angle)
200 Y(SIDE)=radius*SIN(angle)
210 MOVE 0,0
220 PLOT 85,X(SIDE),Y(SIDE)
230 NEXT SIDE
240 MOVE 0,0
250 PLOT 85,radius,0
260 GCOL 0,RND(3)
270 FOR SIDE=1 TO sides
280 FOR line=SIDE TO sides
290 MOVE X(SIDE),Y(SIDE)
300 DRAW X(line),Y(line)
310 NEXT line
320 NEXT SIDE
330 NEXT C
```

DRAW

This program is a simpler version of the SKETCH program on the Introductory Cassette. The main part of the program is between lines 20 and 220, and two procedures are called from here.

Lines 240 to 290 print the instructions at the bottom of the screen.

Lines 320 to 360 limit the graphics area you can draw in, and contain the **DRAW** instruction.

Lines 130 to 200 define which keys on the keyboard are 'drawing' keys, and set the values for X and Y for each key. These values are used later on by line 360.

Note that line 40 turns the cursor off, so when you've finished drawing masterpieces, press **ESCAPE** and type

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

Alternatively, just press **BREAK**.

```
10 REM DRAW
20 MODE1
30 PROCKEY
40 VDU 23,1,0;0;0;0;
50 GCOL 0,1
```

```
 60 VDU 19,1,2,0,0,0
 90 X=640
100 Y=512
110 MOVE X,Y:DRAW X,Y
120 REPEAT
130 L=INKEY(-98)
140 R=INKEY(-67)
150 U=INKEY(-73)
160 D=INKEY(-105)
170 IF L=-1 THEN X=X-4:PROCDRAW(X,Y)
180 IF R=-1 THEN X=X+4:PROCDRAW(X,Y)
190 IF U=-1 THEN Y=Y+4:PROCDRAW(X,Y)
200 IF D=-1 THEN Y=Y-4:PROCDRAW(X,Y)
210 UNTIL FALSE
220 END
230 DEF PROCKEY
240 PRINT TAB(0,25) "Your Drawing keys are:"
250 PRINT TAB(0,26) "Z or z = Left"
260 PRINT TAB(0,27) "X or x = Right"
270 PRINT TAB(0,38) "* or: = Up"
280 PRINT TAB(0,29)"? or / = Down"
290 PRINT TAB(0,30) "Press two keys for a d
iagonal"
300 ENDPROC
310 DEF PROCDRAW(X,Y)
320 IF X<0 THEN X=0
330 IF X>1279 THEN X=1279
340 IF Y<250 THEN Y= 250
350 IF Y>1023 THEN Y=1023
360 DRAW X,Y
370 ENDPROC
```

9 Recording programs on cassette

As you have seen, the Introductory Cassette has a number of programs stored on it. You can record programs you have typed into the Electron onto cassette for future use. This is very useful for transferring other people's programs from their cassette onto yours; for example, you might want to copy one particular program on the Introductory Cassette onto another cassette. Before you make any cassette copies of any programs, be sure that you are free to do so. The company or person who wrote the program may own the copyright to that program, in which case, written permission must first be applied for.

The main thing to remember when you record programs is *where* the program that you've recorded can be found on the cassette, otherwise you will spend a lot of time searching. We strongly advise you to keep a piece of paper with each cassette, and to write down the name of each program and the tape counter position where it begins and ends. Bear in mind that when you record a program, it will record over the top of anything already on the tape; this is useful for erasing old programs you no longer need, but fatal if you record over the top of one you want to keep!

Most short programs will only move the cassette tape counter on 3Ø or 4Ø positions, but try to spread the programs over the length of the cassette. For example, record the first program at ØØØ, the second at 1ØØ, the next at 2ØØ and so on. This will make them easy to find, and will reduce the chances of overlapping recordings. The quickest way to find out if there's a program at a particular point on the cassette is to play it back and listen to the cassette machine if you can. If there is a high pitched whistling noise, it means that a program is just about to start or just finishing. If you hear a screeching noise, you are listening to a program.

One final point – when recording a program at the beginning of a cassette, wind the tape by hand until the clear plastic tape 'leader' is no longer visible.

Saving (recording) a program on cassette

Once you have typed a program into the Electron, then do the following to save it:

Insert the cassette into the cassette recorder.

Set the tape counter to 000 when the tape is fully re-wound.

Type

```
SAVE "MYPROG" RETURN
```

The message

```
RECORD then RETURN
```

appears on the screen.

Fast forward the cassette to the place where you want to record the program – this will be 100, 200 or 300 etc on the tape counter. Note that if you have cassette motor control, you cannot wind the tape unless you have executed a **SAVE**, **LOAD** or ***CAT** command (see below for ***CAT**).

Start recording on the cassette machine, then press **RETURN** on the Electron.

If you want to give up at any time, press **ESCAPE**.

While the program is being saved on the cassette, the name of the program, which is MYPROG appears on the screen along with some numbers. This means everything is going OK. When the computer is finished, the > will reappear, and the tape will stop automatically. If you don't have cassette motor control then the tape will carry on, and you will have to press the STOP button on the cassette recorder after the > reappears.

Checking a recording

To check that the program is really on the cassette use the ***CAT** command described later in this chapter. If the recording went wrong for any reason, then just re-record it.

In the example above, the program was called MYPROG, but you can make up any name you like – as long as it contains ten letters or less.

Loading a program from cassette

To load a program on cassette into the Electron's memory, type

```
LOAD "MYPROG" RETURN
```

The message

```
Searching
```

will appear. Rewind the cassette to just before the beginning of the program, using the tape counter to help you get there.

Check that the volume and tone controls are set correctly. If you are unsure about this, turn to chapter 3.

Press the PLAY button on the cassette recorder.

If the computer finds a program other than the one you asked for, it will display its name on the screen but won't load it. As soon as the computer finds the beginning of your program called MYPROG, the message

```
Loading
```

will appear, and this tells you the computer is now loading that program.

When the program is loaded, the computer will print the > prompt on the screen, and will automatically stop the tape if you have motor control. If you haven't, then press the STOP button.

Now the program is in the computer's memory, type

```
RUN RETURN
```

and the computer executes the program. If you have read the chapter on the Introductory Cassette, you probably remember that we were using a different command to load the tape. This is the **CHAIN** command, and what it does is to tell the computer to first **LOAD** the program and then **RUN** it immediately afterwards. So if you type

```
CHAIN "MYPROG" RETURN
```

this will save you having to type **RUN** after the program has loaded.

LOADING AND SAVING SHOULD NORMALLY BE DONE IN MODE 4, 5 OR 6.

Cataloguing the tape

To find out what programs are on the tape, type

```
*CAT RETURN
```

then play the tape (from the beginning if you want). Better still, keep a record of what is on the tape because cataloguing the tape takes a long time. However, cataloguing the tape also lets the computer verify the information recorded. If there are any errors in the data on the tape, an error message will appear on the screen, and the cataloguing continues.

What the numbers mean

A typical catalogue might look like this

```
MYPROG      00    0084
GAME1       08    088E
GAME2       0A    0ABA
fred        25    2545
```

The program name (or 'filename') is followed by two 'hexadecimal' numbers which give the 'block' number. As described in chapter 4, each program is recorded as a series of 'blocks'. See chapter 6 for an explanation of hexadecimal numbers.

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

Escape

If you want to stop in the middle of a **LOAD**, **CHAIN** or **SAVE**, press **ESCAPE**. You will probably get a

```
Bad Program
```

error appear on the screen. To get rid of this, type

```
NEW RETURN
```

For more information about using cassettes for storing programs, please turn to chapter 26.

1Ø The FUNC key and BASIC keywords

As mentioned in chapter 5 on the Electron keyboard, the **FUNC** key to the left of the keyboard may be used with many of the other keys to print complete BASIC keywords on the screen. For example, if you press **FUNC** L the keyword **LIST** appears. Each key used together with **FUNC** will give a keyword as follows:

A	**AUTO**
B	**RENUMBER**
C	**COLOUR**
D	**DRAW**
E	**ELSE**
F	**FOR**
G	**GOTO**
H	**DEG**
I	**INPUT**
J	**RAD**
K	**CHAIN**
L	**LIST**
M	**MODE**
N	**NEXT**
O	**OLD** ↵
P	**PLOT**
Q	**LOCAL**
R	**RUN** ↵
S	**STEP**
T	**THEN**
U	**UNTIL**
V	**VDU**
W	**RESTORE**
X	**PROC**
Y	**REPEAT**
Z	**END**
,	**LOAD**
.	**SAVE**
/	**PRINT**

OLD and **RUN** incorporate **RETURN**, thereby issuing a direct command. If you press **FUNC** R then the current program (if any) will run.

FUNC may also be used with the numeric keys, which may be programmed to give any string you choose (see chapter 24).

The action of **FUNC** with all the keys can be altered using operating system call ***FX226** and ***FX227** (see Appendix D).

Another way in which to cut down on typing is to use the abbreviations given in the BASIC keyword alphabetical reference section in chapter 25.

11 Variables and expressions

What is a variable?

A variable is piece of memory which is given a name, like Fred or Number or X or Y or virtually anything you want, and this memory is set aside for storing information. It is rather like a box where you and the computer can put useful items of information until they are needed at a later stage. All the computer has to be told is what the box is called, and what kind of information it can expect to find inside. Not only that, but the contents of a box can be changed at any time; so the computer can go to the box to store information, retrieve it, use it, change it, then put it back inside again as many times as you instruct the computer to do so.

There are three types of 'boxes' or variables which the computer can use, and these are used to store three types of information. Briefly, these are:

- A 'real' variable, which can store numbers or fractions, eg 123.654.
- An 'integer' variable, which can store only whole numbers, eg 123.
- A 'string' variable, which can store 'strings' of characters such as words.

Each type is distinguished by the last character of the variable name. A name by itself, like BERT, signifies a real variable, BERT% an integer variable and BERT$ a string variable.

Real variables

Press **BREAK** and type the following program (the line numbers are shown, but you will not need to type them if you are using **AUTO**).

```
>10 PRINT 3+2,3-2,3*2,3/2
>20 A=3
>30 B=2
>40 PRINT A+B,A-B,A*B,A/B
```

If you run this program, you will see that the numbers

```
5          1          6          1.5
5          1          6          1.5
```

are printed on the screen. The first row shows the results of the calculations performed by the **PRINT** instruction. The second row again shows the results, except that this was arrived at by lines 2Ø to 4Ø which use real variables.

Line 2Ø tells the computer that there is a variable in the program called A, and sets its current value to 3.

Line 3Ø tells the computer that there is another variable called B, and its current value is 2. Now that the computer is aware of these two variables, you can tell it to use them in calculations. Thus in line 4Ø, the computer looks for the number stored in each variable, performs the necessary calculations, and the **PRINT** instruction prints the results on the screen just like it did for line 1Ø.

Operators and expressions

Things like 3 + 2, A*B, (FRED – 4)*B are called *expressions.* In general, an expression is a sequence of numbers and variables together with mathematical symbols like +, *, /. These symbols, which are called the 'arithmetic operators', have their normal mathematical meaning, except that in BASIC, * is used for 'multiply' and / for 'divide'.

Here is a list of the arithmetic symbols or 'operators' used in Electron BASIC:

+	addition
–	subtraction
*	multiplication
/	division
^	raise to the power
.	decimal point

For a description of operator precedence, see chapter 12.

Rules for variable names

The rules for naming variables are:

- There must be no spaces in the name.
- The name must start with a letter.
- There must be no punctuation marks in the name and no arithmetic operators. Underline characters may be used.
- The name must not begin with a BASIC keyword (such as **LIST** or **RUN**).

All the following names are acceptable:

```
X = 6.6

SMALL = -30

small = -60

xy = 4*3

height6 = 5/11

William1 = 1066

space_rocket_speed = 25.000
```

Note that capital and small letters are regarded as different by Electron BASIC, so that **SMALL** and **small** are two different variables. Underlines take the place of spaces, which are not allowed.

The following are not acceptable:

```
6teen = 16          (begins with a number)

TOTAL = 77          (begins with TO)

see-saw = 16        (contains a minus sign)

LOW LINE = 3.333    (contains a space)

How! = 1            (contains punctuation mark)
```

A variable does not have to be specified in terms of numbers; it may be specified in terms of other variables, or a mixture of variables and numbers. A statement of the form 'variable = expression' is called an *assignment statement*: it assigns the value of the expression to the variable. For example:

```
X = Y

Monday = Tomorrow

AGE = HEIGHT - 100

TALL = TALL + 1
```

The last assignment of this group is very common. It has the effect of increasing the value of the variable TALL by 1. It is read as 'Add 1 to the number contained in TALL, and store it in TALL again'.

Integer variables

The variables described so far in this chapter are called *real variables*. This means that they can represent both whole numbers *(integers)* and decimal fractions. There are variables called *integer variables* which can be used on the Electron, and these are used for storing only whole numbers. They are signified by the % symbol after the variable name. For example,

```
SCORE% = 20

Hour% = 3600

Z% = -747
```

A% to Z%

The 26 integer variables A% to Z% are called *resident integer variables*, because they are not cleared when the program is **RUN**, or when **NEW** or **BREAK** is used. This means that values can be passed from one program to another. They also have special uses when you come to look at Assembly Language programming (see chapter 29).

Real versus integer variables

The reasons for using integer variables are:

- They occupy slightly less memory than do real variables.
- They are absolutely accurate provided you do not let them get out of range. Real variables are only accurate to nine figures.
- They are much quicker for the computer to process and carry out arithmetic functions.

However:

- Decimal fractions can only be stored in real variables.
- Much larger and much smaller numbers can be stored in real variables. Real numbers can have values of up to approximately 17Ø,ØØØ,ØØØ,ØØØ,ØØØ,ØØØ,ØØØ,ØØØ,ØØØ,ØØØ,ØØØ,ØØØ,ØØØ or $1.7 \times 1Ø^{38}$ (though they are only accurate to the first nine numbers or nine significant figures).

The range and accuracy of real and integer variables are shown in the following table:

	Integer	**Real**
Example	64	1.732
Typical variables	A%	A
Maximum size	2,147,483,647	$1.7 \times 1Ø^{38}$
Accuracy	absolute	9 sig figs
Stored in	32 bits	4Ø bits

DIV and MOD

There are two special arithmetic operators which give integer results. These are called **DIV** and **MOD**.

DIV is an integer division function. It gives the whole number part of a division, for example: 9 **DIV** 2 is 4, 1Ø.5 **DIV** 3 is 3.

When decimal numbers are used, such as in the second example above, the computer truncates the number (meaning that it ignores the decimal part) before it carries out the division: 8.1 **DIV** 2.9 is 4.

MOD stands for modulo, and is used to give the remainder after an integer division. For example: 9 **MOD** 2 is 1, 17 **MOD** 7 is 3.

Once again, decimal numbers are truncated before the division takes place. For example: 16.1 **MOD** 3.8 is 1.

The TIME integer variable

There is also a special integer variable, resident in the computer, which is called TIME. TIME is an elapsed-time clock: it ticks away in hundredths of a second. Every 1/100 of a second its value increases by 1, and it is used for timing programs.

```
10 T%=TIME
20 PRINT TIME -T%
```

will print the time taken to execute one line of program, in hundredths of a second.

TIME may be assigned a starting value, or it can be zeroed, just as any other variable:

```
TIME=0
```

TIME runs continuously for as long as the computer remains switched on. You will understand better how to use it when you look at some of the programs later in the book.

String variables

You have seen that a variable is a name which can be assigned a value either directly or by an assignment statement. The computer will store this value in its memory as a binary number – a series of zeros and ones. Characters are also stored in the computer as binary numbers, and each character has a code. This code is called ASCII, standing for 'American Standard Code for Information Interchange'. If you look in Appendix F, you will see a table of ASCII codes showing all the letters, symbols, and numbers each with their corresponding ASCII code number.

When you use the **PRINT** instruction to put a message on the screen, as for example:

```
PRINT "ASCII"
```

the quotation marks each side of the message tell the computer that what is in between them is a string of characters and not a variable. So each of the characters in the message 'ASCII' is stored as a binary number, corresponding to 65, 83, 67, 73, 73 in decimal, as you can see from the ASCII chart in Appendix F.

There are special variables, called string variables, which hold characters as opposed to numbers. String variables are signified by a $ sign after the variable name. So we can say:

```
A$="ACORN"
fish$ = "TWO COD"
Birthday$ = "Monday 23rd August"
```

It is very important for you to understand how this last assignment is stored. Notice that the string contains a number, 23. Because of the quotation marks this number is not stored as 23 in binary, but as the ASCII code for 2 followed by the ASCII code for 3. This knowledge is very useful when you come to manipulate strings using their code values.

For example:

```
PRINT "23"
```

and

```
PRINT 23
```

both have the same effect.

But

```
PRINT "23*6"
```

and

```
PRINT 23*6
```

show the different ways in which numbers and strings are stored. As you can see from the ASCII table in Appendix F, every number has its own ASCII code.

You can use the computer to find out the ASCII code of a character.

```
PRINT ASC "Q"
```

will give the ASCII value of Q which is 81.

The opposite function is given by

```
PRINT CHR$ 81
```

which converts the ASCII code 81 into its corresponding character which is Q.

Even a space has an ASCII code.

```
PRINT ASC " "
```

gives 32.

And nothing at all (an empty string):

```
PRINT ASC ""
```

gives −1.

This is not an ASCII value, but is conveniently different from all the others as to be easily distinguishable.

The instruction

```
PRINT CHR$ 81
```

has an equivalent which is easier to type:

```
VDU 81
```

is identical, so VDU 81 gives the letter Q.

Commands operating on strings

LEN

String variables may be up to 255 characters long, and there is an instruction **LEN**, which gives the length of a string – the number of characters it contains.

```
PRINT LEN "ABCDEF"
```

will print 6 on the screen.

Similarly,

```
A$="S O S"
PRINT LEN A$
```

will print 5 (because a space is a character).

Linking strings

Two or more strings may be linked together by using the '+' operator, which apart from its arithmetic use, can simply link strings. The following program is an example of this.

```
10 A$ = "I'M"
20 B$ = "LEARNING"
30 C$ = "BASIC"
40 D$ = A$ + B$ + C$
50 PRINT D$
>RUN
I'MLEARNINGBASIC
```

LEFT$, RIGHT$, MID$

Not surprisingly, if the computer can link strings it can also disassemble a string to make smaller ones, using **LEFT$**, **RIGHT$**, and **MID$**.

```
10 A$ = "INEQUITABLE"
20 B$ = LEFT$(A$,2)
30 C$ = RIGHT$(A$,5)
40 D$ = MID$(A$,3,4)
50 PRINT B$
60 PRINT C$
70 PRINT D$
>RUN
IN
TABLE
EQUI
```

Notice how the three functions **LEFT$**, **RIGHT$**, and **MID$** are used:

LEFT$ (A$,2) copies the first two characters of string A$. In the program, these two characters are copied into B$.

RIGHT$ (A$,5) copies the last five characters of string A$.

MID$ (A$,3,4) copies four characters from string A$, beginning at the third character from the left.

VAL, EVAL, STR$

There are three more string operating functions which convert to or from numbers: **VAL**, **EVAL**, and **STR$**.

```
10 X$ = "57/7 * SIN.6"
20 PRINT VAL X$
30 PRINT EVAL X$
```

When you run this program, **VAL** X$ gives the number with which the string X$ begins, in this case 57. If the string does not begin with a number then **VAL** returns the value Ø.

EVAL X$ evaluates the string as if it were a numeric function, giving in this case 4.597Ø3. **EVAL** will also evaluate variables in strings, provided these variables have been assigned earlier in the program.

Sometimes you need to turn a number into a string, and this is done by using the instruction **STR$**.

```
10 A=45 : B=3Ø
20 A$ = STR$(A)
30 B$ = STR$(B)
40 PRINT A + B
50 PRINT A$ + B$
```

INSTR

Another useful string function is **INSTR** (standing for IN STRING) which will compare two strings and tell you whether one of these strings is contained within the other, and at what position.

For example

```
10 A$ = "INEQUITABLE"
20 B$ = "I"
```

```
30 Z = INSTR(A$,B$,2)
40 PRINT Z
```

This program shows that **INSTR** returns the position at which B$ is the same as A$. We can start the **INSTR** comparison at any point along the string.

This program starts the comparison at the second character of string A$, and therefore indicates the second 'I' at position 6. If **INSTR** is used and there is no similarity between the strings, a Ø is given.

STRING$

The last string function, **STRING$**, is used when you want to make a long string which consists of repeated units. For example, if you wish to use a string to print a border made up from *-*-*-*- etc then it is easier to use the **STRING$** function than to type all the characters.

```
10 A$ = "*-"
20 B$ =STRING$(20,A$)
30 PRINT B$
>RUN
*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-
```

The string B$ is made up from 2Ø copies of the string A$.

Comparison table of variables

Finally, here is the complete comparison table for integer, real, and string variables:

	Integer	Real	String
Example	81Ø	1.141	"WORDS"
Typical variables	A%	A	A$
Maximum size	2,147,483,647	$1.7 \times 1Ø^{38}$	255 characters
Accuracy	absolute	9 sig figs	—
Stored in	32 bits	4Ø bits	ASCII values

12 Operator precedence

When a mathematical or logical expression is being evaluated, all the operators (+,*,**DIV** etc) are given a priority of from 1 to 7. Priority 1 operators are those acted upon first, and priority 7 last.

Here is the complete list:

Priority		Operator
1	-	Unary minus
	+	Unary plus
	NOT	Logical NOT
	FN	Functions
	()	Brackets
	?!$	Indirection operators
2	^	Raise to the power
3	*	Multiplication
	/	Division
	DIV	Integer division
	MOD	Integer remainder
4	+	Addition
	-	Subtraction
5	=	Equal to
	<>	Not equal to
	<	Less than
	>	Greater than
	<=	Less than or equal to
	>=	Greater than or equal to
6	**AND**	Logical and bitwise AND
7	**OR**	Logical and bitwise OR
	EOR	Logical and bitwise Exclusive OR

Operators with the same priority are executed left to right, as they appear in the expression. For example, **22 MOD 3/7** is evaluated as **(22 MOD 3)/7**. All priorities may be overridden by using brackets.

13 Arrays

Arrays are groups of variables. An array variable has a name, just as any other, and it also has one or more subscripts. A subscript is a number, and an array variable is numbered according to its position in the array. For example, **A(Ø)** is the first variable in the array named **A**, and **A(1)** is the second variable. The computer must be told how many variables you wish to use in the array, and this is done by using a **DIM** instruction:

```
DIM A(9)
```

allocates space in the computer's memory for ten variables, each called A, but each having a different subscript.

These variables are **A(Ø)**, **A(1)**, **A(2)** . . . **A(8)**, **A(9)**. Each one of these variables may be individually assigned, just like any ordinary variable.

String arrays may also be used.

```
DIM A$(9)
```

allocates space for ten string variables – each of up to 255 characters.

The examples shown above are one dimensional arrays – you can think of them as a line containing a number of variables, subscripted from Ø to 9 in sequence. Two dimensional arrays can be thought of as the printing on the TV screen. Each character printed on the screen is at a particular position from the left, and a particular position from the top.

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

allocates space for nine variables, each called A, and each having two subscripts:

```
A(Ø,Ø) A(1,Ø) A(2,Ø)
A(Ø,1) A(1,1) A(2,1)
A(Ø,2) A(1,2) A(2,2)
```

Arrays may have as many dimensions as you like, may contain as many variables as you like, and may be either real, integer, or string.

One of the invaluable features of an array variable is that its subscript need not be specified as a number; you can use a variable:

```
10 DIM A(9)
20 X = 6
30 A(X) = 27
```

This program will place 27 in **A(6)**. You can even use an array variable as the subscript:

```
10 DIM A(29)
20 X = 6
30 A(X) = 27
40 A(A(X)) = 564.3
```

Any arithmetic expression may be used as a subscript, and if the subscript works out to a number with a decimal point, then the number is truncated to its integer value, ie just the part before the decimal point.

Remember, when using arrays, that if you **DIM** using three subscripts, each variable must be called with three subscripts.

```
10 DIM NAME$(2,2,2)
20 NAME$(0) = "FRED"
```

will not work. You would have to use, say,

```
20 NAME$(0,0,0) = "FRED"
```

Be careful when using arrays – they consume vast amounts of memory, and if you try to use too many variables the computer will say,

```
DIM Space
```

meaning that there isn't enough room in its memory.

14 READ...DATA... RESTORE

These instructions provide you with a way of giving data to a program before it is run. This means that the data can be saved as part of the program (see chapter 9). The variables are placed after **READ**, and each is loaded in turn from the information placed after **DATA**.

```
10 FOR I = 1 TO 4
20 READ Age%,Dog$
30 PRINT "Name: ";Dog$;" Age: ";Age%
40 NEXT I
50 DATA 9, BONZO,3,ROVER
60 DATA 7
70 DATA SPOT,12,HENRY
>RUN
Name: BONZO Age: 9
Name: ROVER Age: 3
Name: SPOT Age: 7
Name: HENRY Age: 12
```

You can see that it doesn't matter how many **DATA** instructions are used provided the types of data match the variable.

RESTORE is used to set the data-pointer to the start of the **DATA**, in this case line 50. It can also be used to set the data-pointer to any line number:

```
10 INPUT "Which dog (1 to 4)", A
20 RESTORE (A+4)*10
30 READ Age%, Dog$
40 PRINT "NAME: ";Dog$;" Age: ";Age%
50 DATA 9,BONZO
60 DATA 3, ROVER
70 DATA 7, SPOT
80 DATA 12, HENRY
>RUN
Which dog (1 to 4)?2
NAME: ROVER Age: 3
```

15 PRINT formatting and INPUT

PRINT formatting

You are already familiar with the **PRINT** statement, and how it is used to put characters on the screen. In this chapter you will find out how to use **PRINT** to position the output on the screen.

Press **BREAK** and then try the following program.

```
10 X = 6
20 PRINT X;X;X
30 PRINT X,X,X
40 PRINT X'X'X
>RUN
          666
          6           6           6
          6
          6
          6
```

From this you can see that

(i) Items in the **PRINT** instruction separated by semicolons are printed one after the other, with no spaces.

(ii) Items separated by commas are tabulated into columns. These columns are ten characters wide and are called fields.

(iii) Items separated by apostrophes are printed on a new line.

Now try the following program:

```
10 A$ = "J"
20 PRINT A4;A$;A4$;A$;A$
30 PRINT A$,A$,A$
40 PRINT A$'A$'A$
>RUN
```

```
J J J
J              J              J
J
J
J
```

This is the same program as before, but using a string variable. Notice that the character is not printed in the same place as the number. Here is another program to demonstrate this:

```
10 X = 6
20 A$ = "J"
30 PRINT X, XX,X'A$,A$,A$
>RUN
             6              6              6
J            J              J
```

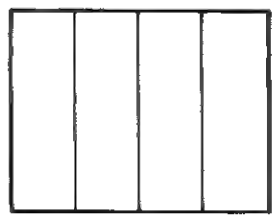
Field No 1 Field No 2 Field No 3

Each field of ten characters width is shown above. As you can see, numbers are printed out to the right of each field, characters to the left. This is done so that numbers line up in the units column, or the least significant decimal.

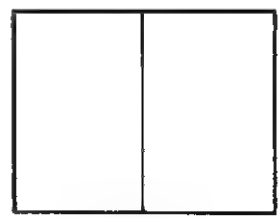
The width of these fields can be altered, as can the number of decimal places to which real numbers are printed. The Electron normally gives each field a width of ten characters. The number of fields across the screen depends upon which mode you are using. There are three different character sizes that are available, and these give either eight fields, four fields, or two fields, each of width ten.



80 character modes
(MODES 0 and 3)



40 character modes
(MODES 1,4,6)



20 character modes
(MODES 2 and 5)

If you now type

```
PRINT 6,9,7/3,57
```

you will see that the 9 and the decimal equivalent of 7/3 have run into each other:

```
 6            92.33333333          57
```

To prevent this happening the field width and/or the number of decimal places can be altered using the integer variable @%.

If you type

```
@% = &2040A
```

and then

```
PRINT 6,9,7/3,57
```

you can see the effect of reducing the number of decimal places to 4.

```
 6.0000     9.0000     2.3333    57.0000
```

The assignment of the variable @% is made up of a number of parts:

& indicates that a hexadecimal number follows. Hexadecimal numbers are explained in chapter 6.

After the & sign, the first number of @% indicates the format of the print field. 2 means that the computer will print a fixed number of decimal places.

The next two figures indicate the number of decimal places which are required, in this case Ø4.

The final two figures give the field width, in this case ØA which is 1Ø in decimal.

So,

```
@%  = &20105
```

will give each number printed to one decimal place, with a field width of five.

Some more points:

(i) The format, the first figure after the **&** symbol, can take three values:

Ø is the normal configuration – the format which the computer uses when it is first switched on.

1 gives numbers in exponent form, that is an integer followed by a power of ten. So **Ø.ØØØ6** would be printed 6E-4. ('Six times ten to the power of minus four'.)

2, as just shown, gives numbers to a fixed number of decimal places.

(ii) When the computer is first switched on, **@% = ØØ9ØA**. This gives nine significant figures and a field width of ten.

(iii) The computer will not print more than ten significant figures. The ten significant figure format is obtained by setting **@%** to, for example, **&ØØAØC**. This will give ten significant figures and a field width of 12. Alternatively, typing **@% = 12** will give the same result, because the number of significant figures is assumed to be ten if it is not specified.

You can make the print instruction convert decimal numbers or variables into hexadecimal by using the ~ character.

```
PRINT ~10
```

will give **A** (decimal 1Ø in hex).

```
PRINT ~LENGTH
```

will print out the contents of the variable **LENGTH** in hex.

The position on the screen at which **PRINT** prints is controllable by the **TAB** instruction.

```
PRINT TAB(16); "J"
```

will print the character J 16 spaces across the screen.

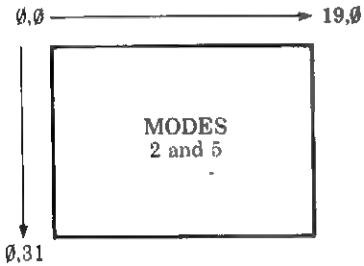
As is usual with functions, the number in the brackets may be a variable, or any arithmetic expression.

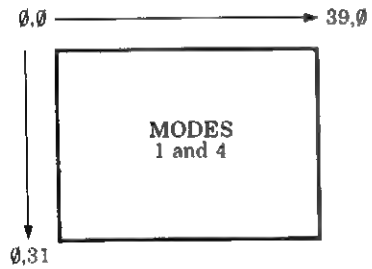
TAB can also be used with two parameters (numbers in the brackets). If you imagine the screen to have coordinates, the number of columns going across the top, and the number of rows down the side, then

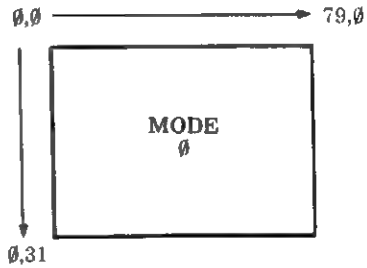
```
PRINT TAB(16,22); "J"
```

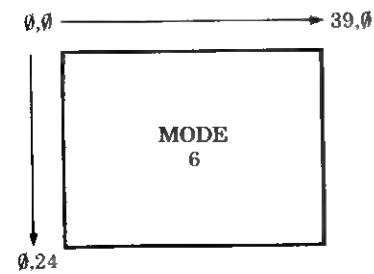
will print the character J 16 spaces across the screen and 22 spaces down.

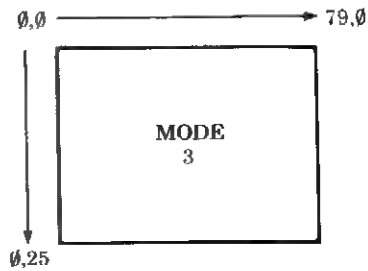
These text coordinates vary depending upon which mode you are using.











TAB instructions have the effect of moving the cursor around the screen.

```
PRINT TAB(0,0)
```

will always move the cursor to the top left of the screen in any mode.

If at any time you wish to turn the cursor off, you can do so by typing

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

It will still exist of course, but it will not be printed on the screen.

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

will return the cursor to the screen once more.

INPUT

So far, the only form of input that you've made to the computer is the typing of commands and programs. Often you will need to give the computer information while the program is running.

```
10 PRINT "GIVE ME A NUMBER AND I'LL DOUBLE
IT"
20 INPUT X
30 PRINT "DOUBLE ";X;" IS ";X*2
>RUN
GIVE ME A NUMBER AND I'LL DOUBLE IT
?16
DOUBLE 16 IS 32
```

When you run this program, line 20 prints a question mark on the screen. This question mark invites you to type in some data. When you press **RETURN** the number that you typed is put in X. If you don't type a number, or you type letters or symbols instead, X becomes zero.

INPUT may also be used with string and integer variables.

```
10 PRINT "WHAT IS YOUR NAME"
20 INPUT A$
30 PRINT "NICE TO MEET YOU, ";A$
```

```
>RUN
WHAT IS YOUR NAME
?DOBBIN
NICE TO MEET YOU, DOBBIN
```

Lines 1Ø of the above two programs have been used to print a message on the screen. This message can be incorporated into the **INPUT** instruction.

```
1Ø INPUT "WHAT IS YOUR NAME ",A$
2Ø PRINT "ARE YOU SURE ABOUT THAT, " A$;"?"
>RUN
WHAT IS YOUR NAME ?EINSTEIN
ARE YOU SURE ABOUT THAT, EINSTEIN?
```

Notice the comma in line 1Ø of this program. It tells the computer to print a question mark when it wants input from the keyboard. If you leave out the comma, the question mark will not be printed. A semi-colon may be used, and has exactly the same effect as the comma.

When the program is running, the **INPUT** instruction requires you to press **RETURN** when you wish to send what you have typed to the computer. Up until **RETURN** is pressed, you can delete all or part of what you have typed using **CTRL** U or the **DELETE** key.

When you are inputting a string, the computer will ignore any leading spaces, and anything after a comma, unless you put the whole string inside quotation marks.

```
1Ø INPUT A$
2Ø PRINT A$
>RUN
?BUS, CAR
BUS
>RUN
?"BUS, CAR"
BUS, CAR
```

Alternatively, you can use **INPUT LINE**, and inverted commas will not be needed.

```
10 INPUT LINE A$
20 PRINT A$
>RUN
?FISH, AND CHIPS
FISH, AND CHIPS
```

Several inputs can be requested at one time.

```
10 INPUT A,B,C$
20 PRINT A,B,C$
>RUN
?20.3, -16, INCHES
      20.3         -16INCHES
```

This time you must use commas to separate each piece of data which you type.

Another way of entering data is to use **GET$**. This reads the key which you press.

```
10 PRINT "PRESS A KEY"
20 A$ = GET$
30 PRINT "THE KEY YOU PRESSED WAS "; A$
```

The program waits at line 20 until you press a key. As soon as you do, the character which that key represents is placed in **A$**.

A similar instruction to **GET$** is **INKEY$**.

```
10 PRINT "PRESS A KEY IN THE NEXT SECOND"
20 A$ = INKEY$100
30 IF A$ = "" THEN PRINT "YOU WERE TOO SLOW"
ELSE PRINT "THE KEY YOU PRESSED WAS ";A$
```

Using this program, line 20 waits one second for you to press a key. If no key is pressed in that time then the program moves on. The **INKEY$** instruction has a number after it which is hundredths of a second. The larger the number, the longer the computer will wait for you to press a key. If you don't press a key in time, **INKEY$** will give the value 1.

Line 3Ø of this program shows an **IF** statement. **IF** statements are discussed in detail in chapter 16.

There are two more input instructions, **GET** and **INKEY**. These are exactly the same in operation as **GET$** and **INKEY$**, but their effect is to give the ASCII code of the key which you press. **GET** and **INKEY** give a number, and must therefore be assigned to number variables.

```
1Ø  X = GET
2Ø  Y = INKEY5ØØ
```

If you use either **INKEY** or **INKEY$** with its time delay set to Ø, the computer will not wait for you to press a key, but will merely glance at the keyboard to see if any key has already been pressed. This is useful when you come to write games, as the rest of the program is not held up waiting for a key to be pressed.

It is important to know that **INKEY**, **INKEY$**, **GET** and **GET$** will notice not just the key you happen to be pressing at the time, but also any key you have pressed since the last time the computer asked for input. Every time you press a key it is placed in the keyboard buffer (a buffer is a short-term memory), and it is this buffer which **GET** and the others actually look at.

However, if you want to ignore any keys previously pressed, and just look at the keyboard directly, there's a version of **INKEY** to do this. This is done by giving **INKEY** a negative number, each key having a corresponding value which you can use. For example, **INKEY(–7Ø)** only operates when the J key is pressed. This use of **INKEY** is discussed in more detail in chapter 25, including a table of all the key codes.

16 Conditional and loop instructions

Programs and parts of programs can be made to execute over and over again either continuously, or a specified number of times. The instructions you put in your program to make this happen are called LOOP instructions.

The FOR . . . NEXT loop

The most common type of loop is **FOR . . . NEXT**, which uses a variable to count the number of repetitions required.

```
10 FOR N = 1 TO 6
20 PRINT N
30 NEXT N
>RUN
          1
          2
          3
          4
          5
          6
```

In this program, N is printed at each pass through the loop. N is called the control variable.

You can start the control variable at any number you choose, and you may alter the amount by which it changes on each pass, the step size.

```
10 FOR N = 7 TO 11 STEP 1.6
20 FOR J = 20 TO 10 STEP -5
30 PRINT N,J
40 NEXT J
50 NEXT N
>RUN
          7          20
          7          15
          7          10
        8.6          20
        8.6          15
```

```
 8.6          10
10.2          20
10.2          15
10.2          10
```

This program shows that you can use decimal step sizes, or negative step sizes. You may start the control variable at any value; and you may use **FOR . . . NEXT** loops within each other. This is called nesting, and you can nest as many loops as you wish.

Type

```
LISTO2
```

followed by **RETURN**, and then **LIST** the program again.

```
10 FOR N = 7 TO 11 STEP 1.6
20   FOR J = 20 TO 10 STEP -5
30     PRINT N,J
40     NEXT J
50   NEXT N
```

Each loop is shown indented from the previous one.

LISTO is a list option instruction and can take any number from Ø to 7.

LISTOØ is the normal setting – it lists exactly what the computer has stored in memory.

LISTO1 lists the program with an extra space after each line number.

LISTO2 lists the program with indents on **FOR . . . NEXT** loops.

LISTO4 lists the program with indents on **REPEAT . . . UNTIL** loops.

These effects may be obtained in any combination by adding the numbers together, so **LISTO3** would give extra spaces after line numbers and indented **FOR . . . NEXT** loops.

Here are some further points on the use of **FOR . . . NEXT** loops.

(i) You do not need to specify the control variable to which **NEXT** refers. The following program will work exactly the same as the one above.

```
10 FOR N = 7 TO 11 STEP 1.6
20 FOR J = 20 TO 10 STEP -5
30 PRINT N,J
40 NEXT
50 NEXT
```

The computer assumes that **NEXT** applies to the loop it is in at the present moment.

If you do put the variable names after **NEXT**, but get them mixed up, then this is what happens.

```
10 FOR N = 7 TO 11 STEP 1.6
20 FOR J = 20 TO 10 STEP -5
30 PRINT N,J
40 NEXT N
50 NEXT J
>RUN
          7          20
        8.6          20
       10.2          20


No FOR at line 50
```

The computer starts to execute the N loop before the J loop, and when it reaches line 50 it cannot find the **FOR** to go with **NEXT J**. Loops must be nested one within another; they must not cross.

(ii) The number of **FOR**s, and the number of **NEXT**s must be the same.

The following program does not give an error, but it is left hanging in mid-air.

```
10 FOR N = 7 TO 11 STEP 1.6
20 FOR J = 20 TO 10 STEP -5
30 PRINT N,J
40 NEXT
>RUN
          7          20
          7          15
          7          10
```

If you do this, you will run into trouble.

(iii) You must never jump out of a **FOR . . . NEXT** loop using **GOTO**. As in (ii) above this will often not result in an error, but the program will be impossible to follow.

(iv) The stop condition for a loop is that, for a positive step size, the control variable is greater than the terminating value; for a negative step size, the control variable is less than the terminating value. However, all loops will be executed at least once.

```
10 FOR NUMBER = 6 TO 0
20 PRINT NUMBER
30 NEXT
>RUN
           6
```

When the loop has been completed, the control variable moves on an extra step, so the above program will end up with **NUMBER** equal to 7. Here is a program to show this:

```
10 FOR Size = 100 TO 103 STEP 1.5
20 PRINT "INSIDE LOOP, Size = ";Size
30 NEXT
40 PRINT "OUTSIDE LOOP, Size = ";Size
>RUN
INSIDE LOOP, Size = 100
INSIDE LOOP, Size = 101.5
INSIDE LOOP, Size = 103
OUTSIDE LOOP, Size = 104.5
```

(v) **FOR . . . NEXT** loops are used when you wish to go around a loop a fixed number of times. There may be as many lines as you like between the **FOR** and its corresponding **NEXT**, and control variables need not be directly assigned with numbers. They can be assigned with any arithmetic expression, containing variables or other functions.

```
10 MODE 5
20 FOR angle = 0 TO 2*PI STEP .1
30 PLOT 69,649 + 440*SIN(angle), 512 + 400*
COS(angle)
40 NEXT
```

The REPEAT . . . UNTIL loop

Another very useful loop is **REPEAT . . . UNTIL**, which waits until a condition is fulfilled before coming out of the loop.

```
10 INPUT'"This program turns decimals into
fractions"'"Give me a decimal: "decimal
20 numerator% = 1: denominator% = 1
30 PRINT "Program running"
40 REPEAT
50 fraction = numerator%/denominator%
60 IF fraction>decimal THEN denominator% =
denominator%+1
70 IF fraction<decimal THEN numerator% = nu
merator%+1
80 UNTIL fraction = decimal
90 PRINT;decimal;"  is equal to  ";numerato
r%;"/"denominator%'"Program end"
```

This program asks you to input a decimal. It then prints out the fractional equivalent. (Don't input too complicated a decimal or the program will run for hours.) Lines 50 and 60 are repeated until the condition at line 80 is fulfilled. In this example, the condition is that fraction = decimal.

Line 80 is called a conditional statement, and the result of a conditional statement may either be **TRUE** or **FALSE**. In the example shown above, the statement becomes **TRUE** when fraction is equal to decimal, so the program loop is repeated only whilst the conditional statement is **FALSE**. Of course, the computer doesn't understand **TRUE** and **FALSE**, so it assigns numeric values to these conditions:

0 for **FALSE**, –1 for **TRUE**.

There are a number of logical operators which can be used in conditional statements:

A = B	True when A is equal to B
A < B	True when A is less than B
A > B	True when A is greater than B
A<= B	True when A is less than or equal to B
A>= B	True when A is greater than or equal to B
A< > B	True when A is not equal to B
NOT A	True when A is false

TRUE	True always
FALSE	False always
A AND B	True if both A and B are true
A OR B	True if either A or B is true, or if both are true
A EOR B	True if either A or B is true, but false if both are true

There is more about logic operations in the next section on the **IF** statement.

REPEAT . . . **UNTIL** is easily followed and understood by other people who read your programs, and should be used in preference to **GOTO**.

IF . . . THEN . . . ELSE

The **IF** statement enables the computer to make a choice about something.

```
10 REPEAT
20 A$ = GET$
30 IF A$ = "Y" THEN PRINT "YES"; ELSE PRINT
 A$;
40 UNTIL FALSE
```

This program turns the Y key into a YES-button. Line 30 contains a conditional statement. If the condition is true then the computer obeys whatever comes after **THEN**. If the statement is false then the computer carries out whatever comes after **ELSE**.

IF statements can carry out more than one instruction, if these instructions are placed on the same line and are separated by colons:

```
10 REPEAT
20 A$ = GET$
30 IF A$ = "Y" THEN FOR I = 1 TO 6 : PRINT
"YES";: NEXT ELSE PRINT A$;
40 UNTIL FALSE
```

and now you will get six **YES**s when you type Y.

These multi-statement lines are not restricted to the **IF** statement. Any line in a program may carry out more than one instruction if each is separated by a colon. However, it is generally better to use a procedure

rather than fill an **IF** statement full of colons. Procedures are explained in chapter 17.

IF statements may ask for a complex condition, using the logical operators **AND**, **OR**, and **EOR**.

```
10 REPEAT
20 A$ = GET$
30 B$ = GET$
40 IF A$ = "Y" AND B$ = "Y" THEN PRINT "YES
";: ELSE PRINT A$;B$
50 UNTIL FALSE
```

This program will only print **YES** if you type two Ys in succession.

The **ELSE** part of the statement is optional, and may be omitted.

Alternatively you can extend the **IF** statement by chaining it:

```
10 REPEAT
20 A$ = GET$
30 IF A$ = "Y" THEN PRINT "YES" ELSE IF A$
= "N" THEN PRINT "NO" ELSE PRINT "MAYBE"
40 UNTIL FALSE
```

This program demonstrates the use of one **IF** statement after another . . .

Using the **IF** statement you can now find out some more about how the computer deals with **TRUE** and **FALSE**.

```
10 X=8>6
20 Y=6>8
30 PRINT X,Y
>RUN
         -1           0
```

Because 8 is greater than 6, $8 > 6$ is **TRUE**, so X is -1. $6 > 8$ is **FALSE**, so Y is Ø.

```
10 REPEAT
20 INPUT X
```

```
30 IF X THEN PRINT;X; " IS TRUE" ELSE PRINT
;X; " IS FALSE"
40 UNTIL 0
```

This program allows you to enter numbers, and to see whether the computer treats them as **TRUE** or **FALSE**. You will see that only Ø is treated as **FALSE**, all other values being **TRUE**.

The above program can be rewritten:

```
10 REPEAT
20 INPUT X
30 IF X THEN PRINT ;X;" IS TRUE" ELSE PRINT
;X;" IS FALSE"
40 UNTIL FALSE
```

which has exactly the same effect.

Another important use of **IF** is with strings and string variables. For example, you might find this in the middle of a program

```
100 REM The answer should be 560
110 INPUT "SO WHAT'S THE ANSWER THEN?"'X
120 IF X = 560 THEN A$ = "YES" ELSE A$ = "N
O"
130 IF A$ = "YES" THEN PRINT A$;" WELL DONE
" ELSE PRINT A$;" TRY AGAIN"
```

This will test to see what string A$ contains, and will print one of two messages accordingly.

Less than and greater than can also be used:

```
10 A$ = "HELLO"
20 IF A$ < "GOODBYE" THEN...
```

This will be true, because the **IF** statement compares the ASCII values of the first character in each string. If the first two characters are the same, then the next two characters are compared, and so on. So 'MELON' is less than 'MELTED'. This is very useful for putting strings into alphabetical order, but it does not work if the strings are a mixture of upper and lower case letters.

The following operators may be used with strings:

=	the same as
<>	not equal to
<	less than
>	greater than
<=	less than, or equal to
>=	greater than, or equal to

17 Procedures

Using procedures allows you to split up virtually any program you are going to write into a main program, followed by a number of 'mini-programs' (procedures), which can be called from the main program by a single statement.

A procedure is simply a collection of numbered BASIC statements which you write in order to perform a particular task. This collection of statements looks just like part of an ordinary program, but the differences are that the first line contains the name of the procedure (which is decided by you), and the last line contains a BASIC word to signify the end of the procedure. When the computer encounters the end, it then returns to the main program and carries on.

The rules for using procedures are very simple. A procedure is called from the main program by the BASIC word **PROC** followed immediately by the procedure's name. The name can be anything you like, but there must be no spaces in it. For example:

```
PROCnewline
PROCwait_a_second
PROCDRAWPICTURE
```

Note the underline character in the second example which helps 'space out' the name.

A procedure name should reflect the function of the procedure to which it applies. If you merely name your procedures **PROCA**, **PROCB**, **PROCC**, for example, then no one will understand what they do without having to work through each one. So if you have a procedure which converts feet into metres, then call it **PROCfeet_to_metres**. It is best to use lower case names for procedures so that they distinguish themselves from the **PROC**.

To define a procedure, you simply type a line number, followed by **DEF**, followed by the procedure name. It is a good idea to start defining your procedures at a fairly high line number, say 1000.

```
1000 DEF PROCwait_a_second
1100 NOW = TIME
1200 REPEAT UNTIL TIME-NOW>=100
1300 ENDPROC
```

This procedure will do as its name suggests.

ALL PROCEDURE DEFINITIONS MUST END WITH **ENDPROC**.

When you want the computer to carry out the instructions in the procedure, you have to call it by name:

```
70 PROCwait_a_second
```

or

```
120 IF INKEY$10 = "W" THEN PROCwait_a_secon
d
```

You may have as many procedures in your program as you like, and usually the more the better.

THERE MUST ALWAYS BE AN **END** INSTRUCTION BETWEEN THE END OF THE MAIN PROGRAM AND THE PROCEDURE DEFINITION.

For example

```
10 REM Sample program
20 FOR X = 0 TO 29
30 PRINT TAB(10,10);"COUNTING..."X" Seconds
"
40 PROCwait_a_second
50 NEXT
60 PRINT TAB(10,10);"Half a minute up!
  "
70 END
1000 DEF PROCwait_a_second
1100 NOW = TIME
1200 REPEAT UNTIL TIME-NOW>=100
1300 ENDPROC
```

There is a program on the Introductory Cassette to illustrate the use of procedures, and also give you some fun. Load this program which is

called 'BUGZAP' into your Electron first. See chapter 4 for instructions on how to load a program.

When you **LIST** a long program obviously you cannot see all of the lines on the screen at the same time. Using **LIST** with specified line numbers is one way around this, but another is to put the computer into paged mode. This is done by pressing **CTRL** N. (No **RETURN** is required.) If you now use **LIST**, the program will be listed until the screen is full. When you want to look at the next part just press **SHIFT** and another screen full will appear. If you want to change a line number, you must press **ESCAPE**. To get the computer out of paged mode, type **CTRL** O.

Look at just one procedure from this program:

```
520DEF PROCinfo
530CLS
540PRINT''''''"Welcome to the game of Bugzap
!"''''
550PRINT"The object of the game is to use y
our"
560PRINT"laser gun to zap the descending bu
g"
570PRINT"before it lands or bombs you."
580PRINT'"Your score increases every time y
ou"
590PRINT"zap the bug, with more points bein
g"
600PRINT"given the lower the bug is; it wil
l be"
610PRINT"displayed when you are killed."
620PRINT'"The controls are:"'
630PRINT"Z      =  left"
640PRINT"X      =  right"
650PRINT"SPACE =  fire"
660PRINT''"Pressing the ESCAPE key will tak
e you"
670PRINT"to the end of the program."
680PRINTTAB(5,31)"Press SPACE to start the
game";
690REPEAT: UNTIL GET$=" "
700ENDPROC
```

This procedure is called from line 90 of the main program.

```
90PROCinfo
```

Line 520 is the start of the definition.

Line 530 clears the screen.

Lines 540 to 680 print the introduction and instructions about the BUGZAP game which you see when you run the program.

Line 690 is an example of putting two separate BASIC statements after one line number by separating them with a colon. The purpose of line 690 is to wait until the space bar is pressed: **GET$ = ""**. When the space bar is pressed, line 700 is executed.

Line 700 signifies the end of the procedure, and the computer goes back to the main program to the line immediately after the procedure call (**90PROCinfo**), which is line 100.

Here is one of the procedures from the 'MARSLANDER' program – also on the Introductory Cassette.

```
860DEF PROCrocket(direction%)
870REM If there is any fuel then fire rocke
t-motor and make sound
880IF fuel% THEN fuel%=fuel%-1 ELSE ENDPROC
890IF fuel%=29 THEN SOUND 1,-10,60,10 ELSE
SOUND 0,-1,5,2
900ON direction% GOTO 910,920,930,940
910VY%=VY%- 5:ENDPROC
920VX%=VX%+10:ENDPROC
930VY%=VY%+15:ENDPROC
940VX%=VX%-10:ENDPROC
```

This procedure alters the speed of the spacecraft according to the direction in which it is pointing, which is given by the variable direction%.

This procedure is called from line 250.

```
250IF INKEY(-99) THEN PROCrocket(Z%)
```

Z% is an integer variable which is used by the program to give the attitude of the spaceship.

When the computer reaches line 25Ø it tests to see if the space bar is pressed. If it is, the computer then places the contents of Z% into direction%.

The variable Z%, and hence the parameter direction%, can be anything from 1 to 4, where 1 indicates the capsule pointing up, 2 to the right, 3 down, and 4 to the left. These positions are represented by characters 224 to 227 which are user-definable.

Line 88Ø checks to see whether there is any fuel left. The variable fuel% will be FALSE when it is zero and the procedure will end.

If it is TRUE one unit of fuel is deducted by decrementing its contents by 1. Line 89Ø makes either a 'beep' (fuel is low), or a rocket motor sound (fuel is not low).

SOUND is explained in chapter 22.

Line 9ØØ uses **ON . . . GOTO**, to act according to the direction of the spacecraft. The parameter direction% now contains the value given to it by Z%. If the spacecraft is pointing up, direction% is 1 and execution continues at line 91Ø.

Line 91Ø decreases the vertical speed of the capsule. (VY% is the vertical speed measured positive in a downward direction; VX% is the horizontal speed measured positive in a left-to-right direction.) If the capsule is pointing to the left line 9ØØ passes execution to line 92Ø which increases the horizontal speed of the capsule.

Lines 93Ø and 94Ø increase the vertical speed and decrease the horizontal speed respectively. After any one of these lines (91Ø to 94Ø) has been executed, the procedure ends.

Using parameters in procedures

Using the above example, Z% and direction% are termed parameters. The idea behind using parameters is that they are more efficient than global variables. A global variable is one which is accessible throughout the whole program, and may be altered or re-assigned at any line number.

Once a global variable such as Z% has been passed to the procedure as a parameter, the variable which takes its place, direction%, is only known to that procedure. Outside **PROCspaceship** you can ask the computer to

```
252 PRINT direction%
```

and it will give an error because the variable direction% does not exist in that part of the program. Global variables which are passed to the procedure are called the actual parameters, and the variables within the procedure are called formal parameters.

A procedure may be defined with only one parameter, or it may be defined with lots of parameters. But a procedure must always be called with the correct number of parameters. **PROCspaceship**, starting at line 75Ø, has three parameters.

So you could not call **PROCspaceship(X%,Y)**.

Parameters may be integer, real, or string. If a string variable is used as a formal parameter then it must have either a string or a string variable passed to it. Real and integer parameters may be passed to one another and interchanged freely, but remember that the fraction part of a real variable will be lost when assigned to an integer variable.

The idea of a variable being defined only within a certain section of a program is commonplace in a lot of computer languages, but unusual in BASIC. Electron BASIC allows you to declare any variable as *local* to a procedure or function (functions are discussed in chapter 19). A local variable may even have the same name as a global variable in the same program, but will lead a separate existence.

For example:

```
10 FOR I = 1 TO 3
20 PROClocal(I)
30 PRINT "OUT OF PROCEDURE I = ";I
40 NEXT I
50 END
60 DEF PROClocal(J)
70 LOCAL I
80 I = J
90 I = I*10
100 PRINT "IN PROCEDURE I = ";I
110 ENDPROC
>RUN
IN PROCEDURE I = 10
OUT OF PROCEDURE I = 1
IN PROCEDURE I = 20
OUT OF PROCEDURE I = 2
```

```
IN PROCEDURE I = 30
OUT OF PROCEDURE I = 3
```

Notice line 50 which says **END**. Because procedures are usually defined at the end of a program, you sometimes need to stop the execution after all the calls have been made. The program will terminate when the computer reaches the instruction **END**.

There is still another way to use procedures, and that is *recursively*. A recursive procedure is one which calls itself from within its own definition.

```
10 answer = 1
20 INPUT X
30 PROCfactorial(X)
40 PRINT answer
50 END
60 DEF PROCfactorial(N)
70 answer = answer*N
80 IF N > 1 THEN PROCfactorial(N-1)
90 ENDPROC
```

This is a recursive procedure to find the factorial of a number. Check through the logic of it in your head to see that it works. Recursive procedures are very useful in certain circumstances, but they consume memory very quickly.

18 GOTO and GOSUB

There are four more instructions in Electron BASIC which can be used to tell the computer to continue executing the program at specified points.

These are:

GOTO
GOSUB . . . RETURN
ON . . . GOTO
ON . . . GOSUB

GOTO

The simplest of these instructions is **GOTO**.

```
10 PRINT "SCREENFUL"
20 GOTO 10
```

Each time the computer executes line 20 it is sent back to line 10 once again. This program never ends: it is a continuous loop. To stop the program you may press either **ESCAPE** or **BREAK**. If you press **ESCAPE** a message is printed giving the line number at which execution ceased.

GOTO instructions may send control of the program either forwards or backwards, but you must be careful not to use too many **GOTO** loops – they soon become impossible to follow, and very difficult to correct when a program does not function as you wish it to. It is far better to use procedures or **REPEAT . . . UNTIL** statements where possible.

GOSUB . . . RETURN

GOSUB stands for 'Go To Subroutine', and is really just a variation of **GOTO**. It is strongly advised that you use the more readable and more flexible procedure instead of **GOSUB**. It is used when a particular routine is used several times in different parts of the same program, for example to read a key.

It is most useful with **IF** statements.

Here is a game which requires you to put a set of numbers in sequence. The **GOSUB** routine is called from various parts of the program, and has the effect of swapping the numbers around according to which key you press.

```
10 REM SWAP-ROUND
20 MODE 6
30 VDU23,1,0;0;0;0;
40 answer$="123456789"
50 number$=answer$
60 INPUT TAB(8,16)"Difficulty level",level
70 FOR I=1 TO level
80 position=RND(8)+1
90 GOSUB 210
100 NEXT
110 CLS:PRINT TAB(15,10);number$
120 PRINT TAB(6,16)"Press a key between 2
and 9"
130 REPEAT
140 position=VAL GET$
150 IF position <2 OR position>9 THEN GOTO
140
160 GOSUB 210
170 PRINT TAB(15,10)number$
180 UNTIL number$=answer$
190 PRINT TAB(6,16);SPC(9);"Well done";SPC(
11);"END"
200 END
210 temporary$=""
220 FOR J=position TO 1 STEP-1
230 temporary$=temporary$+MID$(number$,J,1)
240 NEXT
250 number$=temporary$+MID$(number$,positio
n+1)
260 RETURN
```

As you can see, **GOSUB** differs from **GOTO** in that the program flow must always **RETURN** to the position following the subroutine call.

Just one point about **GOSUB**.

As with **FOR . . . NEXT**, you should not jump out of a subroutine by using **GOTO**. If the computer keeps being told to **GOSUB**, without ever encountering a **RETURN**, it will soon use up its memory.

ON . . . GOTO, ON . . . GOSUB

An instruction such as

```
ON N GOTO 100,200,70,260
```

means that the computer checks on the value of N, and then 'jumps' to the Nth line number in the list.

So, if N = 1, the program 'jumps' to line 100; if N = 2 to line 200; if N = 3 to line 70; and if N = 4 to line 260.

ON . . . GOSUB works in exactly the same way.

19 Functions

Functions are similar to procedures, but they have only one purpose – to give a single result. The easiest way to understand a function is to describe some of the computer's own. It has lots – the trigonometric functions such as **SIN**, **TAN** and **COS**. One of the most useful functions is **RND**, which supplies random numbers. It is usually used with a parameter, and gives a random integer between 1 and the value of the parameter. So, **RND(50)** will pick a random number between and possibly including, 1 and 50. When you type **X = RND(4)**, you know that the result of the function **RND** will be placed in **X**. The **RND** function is described in more detail in chapter 25.

A function can be used with any number of parameters, both string and numeric. Here is a function to determine the mass of a sphere:

```
100 DEF FNmass_of_sphere(radius,density)
110 = 4/3*PI*radius^3*density
```

Here's another example of using a function in a program

```
5 CLS
10 REM Discount calculator
20 PRINT '''''"This program calculates the
 following discounts:"
30 PRINT ''"20% on £100 or less"
40 PRINT ''"30% on £101 to `200"
50 PRINT ''"50% on anything over £200"
60 INPUT ''''"Enter the sum    £" Y
70 PRINT ''''"Final sum with discount is
£";FN_discount(Y)
80 END
100 DEF FN_discount(SUM)
110 IF SUM <= 100 THEN =SUM - (20*SUM/100)
120 IF SUM > 100 AND SUM <= 200 THEN =SUM -
 (30*SUM/100)
130 IF SUM > 200 THEN =SUM - (50*SUM/100)
```

The main program starts at line 5 and ends at line 80.

Line 5 clears the screen, and lines 2Ø to 5Ø print instructions on the screen.

Line 6Ø prints a request for you to enter an amount, waits for you to do so, and puts the value into variable Y.

Line 7Ø prints a message, and calls a function called **FN__discount(Y)**. The value in Y is passed to the function's parameter, (which is the 'actual' parameter).

Line 1ØØ starts the definition of the function, and passes the parameter value to a 'formal' parameter called **SUM**.

Line 11Ø contains a conditional statement. If the value of **SUM** is 1ØØ or less, then the function returns the result given by **SUM – (2Ø*SUM/1ØØ)**. If the value of **SUM** is more than 1ØØ, then the execution of line 11Ø stops before working out the **SUM – (2Ø*SUM/1ØØ)**, and line 12Ø has a go – and so on.

Notice the underline character in **FN__discount**. This helps to make the function's name more readable.

2Ø Graphics

Introduction

This chapter deals with the **VDU** software – anything to do with how things are put on to the screen (ie the television or monitor). What 'modes' are and why they are there is covered first, followed by a section on writing text and then details on the graphics routines. Lastly the palette is covered. All the individual **VDU** commands are listed for reference in the next chapter.

Modes – what are they and why?

The screen displays things in any one of seven modes, labelled from **MODE Ø** up to **MODE 6**. To change mode is easy – just type **MODE** followed by the mode number you want. For example

```
MODE 2 RETURN
```

changes the display to mode 2. As with all **VDU** commands, it can be used as a line in a program, and as it is a good idea to make sure your program starts off in the right mode, have its first line looking something like this:

```
10 MODE 1
```

Changing mode changes four things:

- The number of characters you can get on the screen.
- The number of pixels (dots) the graphics can display (and hence the resolution of the graphics).
- The number of colours available at any one time on the screen.
- The amount of memory left for programs.

A table giving details of these is listed below.

Mode	No of characters	No of graphics pixels	No of colours	Memory used
Ø	8Ø × 32	64Ø × 256	2	2ØK
1	4Ø × 32	32Ø × 256	4	2ØK
2	2Ø × 32	16Ø × 256	16	2ØK
3	8Ø × 25	(text only)	2	16K
4	4Ø × 32	32Ø × 256	2	1ØK
5	2Ø × 32	16Ø × 256	4	1ØK
6	4Ø × 25	(text only)	2	8K

If you don't understand the 'memory used' column then don't worry – basically the more detail and colours available in the mode, the more memory the screen uses and the less there is available for programs. The word 'colour' is used rather loosely to include the flashing colour effects.

Try the different modes out to see the differences. Modes 3 and 6 are for text only – no graphics can be done in these modes (nothing will actually go wrong – it just won't appear).

Why have modes? Different programs have different requirements – some just need simple text output and mode 6 then leaves free as much memory as possible for the program. Others, such as games, need lots of colours and graphics detail. The modes available give a good range across this spectrum.

Writing text

The COLOUR command and text windows

When a letter is written to the screen it has foreground and background colours – the colours of the ink and the paper. When the machine is turned on, it is always white foreground on black background. Colours (or more strictly logical colours – see the section on the palette) are labelled from Ø upwards. To take a definite mode for simplicity, mode 1 has four colours labelled from Ø to 3. Try the following:

```
MODE 1 RETURN
```

This puts you in mode 1 with white text on black background.

```
COLOUR 1 RETURN
```

This sets the foreground colour to number 1 (red). Text after this command is red on black.

```
COLOUR 130 RETURN
```

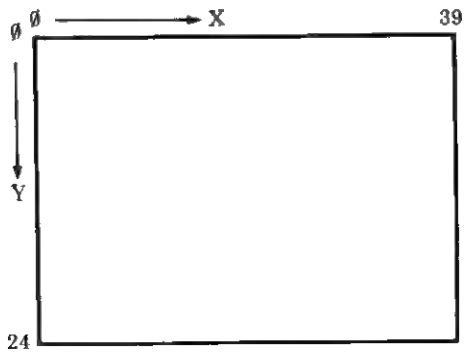
This sets the background colour to number 2 (yellow). Text after this is now red on yellow, and clearing the screen with **CLS** makes the entire screen yellow. Why 13Ø? Because to change the background instead of the foreground colour you must add 128 to the colour number. Thus, to get background colour 2 (above), add 128 to give 13Ø.

Changing mode resets the colours to white on black. As said before, any **VDU** commands (including **COLOUR**, **GCOL**, **MOVE**, **DRAW** etc) can be either typed straight (as a 'direct command') or used as part of a BASIC program.

Addresses on the text screen

Each letter position has its own address in the usual columns and rows format. The column numbering is from left to right starting from column Ø and the rows, as for all **VDU**s, are labelled from the top (row Ø) downwards. How many rows and columns there are depends on the mode – the drawing below shows the labelling for mode 6.

The text screen for mode 6



The cursor may be positioned to any part of the screen with the **TAB(X,Y)** command, thus the following program prints out a diagonal line of As.

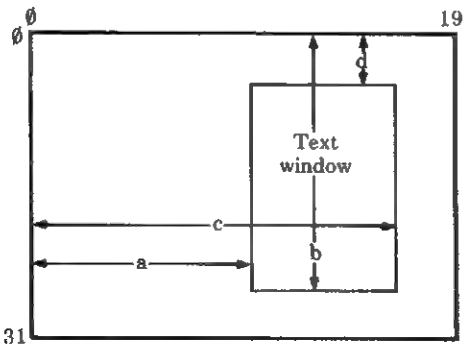
```
10 MODE 1
20 FOR I%=0 TO 20
30 REM The next line positions the text cur
sor to the position col.=I%+5 row=I%
40 REM  and prints the letter A at this pos
ition
50 PRINT TAB(I%+5,I%);"A"
60 NEXT I%
70 END
```

Text windows

Normally, the text may appear anywhere on the screen. However a text window may be set, which allows the text to appear only inside the window. To do this, the **VDU 28** command is used as follows:

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

where a,b is the bottom left and c,d the top right position inside the window (see the drawing below).



Nothing outside the text window is affected by text commands, such as screen clearing, scrolling, cursor positioning etc. Note that the **TAB(X,Y)** measures from the position of the top left of the current window. Try the following program

```
10 MODE 1
20 REM Set up a text window only 6 characte
rs square
30 VDU 28,5,10,10,5
40 REM Change the background colour to colo
ur 1 (red)
50 COLOUR 129
60 REM Now clear the text screen to red to
see where it is
70 CLS
80 REM Demonstrate scrolling
90 FOR I%=1 TO 20 : PRINT I% : NEXT I%
100 REM lastly, show position of character
(2,2) relative to text window
110 PRINT TAB(2,2);"*"
120 END
```

Both text and graphics windows are removed by **VDU 26**.

Defining your own characters

Each character is an 8 by 8 matrix of dots (pixels). All the normal letters, numbers and punctuation marks are defined, but it is possible to define your own. 256 bytes of RAM are set aside for the definitions of characters whose codes are from 224 to 255. Character definitions are entered thus:

VDU 23,CODE,L1,L2,L3,L4,L5,L6,L7,L8

where CODE is the code of the character to be defined (it is then printed using either **VDU CODE** or **PRINT CHR$(CODE);**) and

L1 is the bit pattern of the top row
L2 is the bit pattern of the second row from top, and so on until . . .
L8 is the bit pattern of the bottom row.

What is a bit pattern? Each dot in any one row is given a number, and the bit pattern is the sum of the numbers corresponding to those bits in foreground. These numbers, labelling the bits from left to right, are 128 (for the leftmost pixel), 64, 32, 16, 8, 4, 2, 1 (for the rightmost pixel). Specific examples are easiest to understand.

The space character obviously has no foreground, thus all the bit patterns are zero, so to assign the space character to the code of 224, the command

```
VDU 23,224,0,0,0,0,0,0,0,0
```

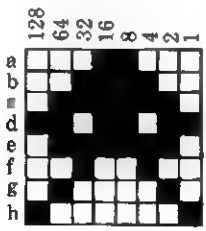
would be used. To define a large X, the top line has the left and rightmost pixels set only, thus L1=128+1=129. The next line has the second from left and the second from right pixels set, thus L2=64+2=66. Similarly, L3=32+4=36 and L4=16+8=24. The fifth through to eighth rows are the mirror image of the first four, so to define the character 225 as an X, type the following line:

```
VDU 23,225,129,66,36,24,24,36,66,129 RETURN
```

To display the character, type **VDU 225 RETURN**

All the characters from 32 to 255 may be defined, but to define those outside the codes 224-255 it is necessary to allocate more memory for the fount. This is called 'exploding the fount' and is done via **FX** call number 20.

Here is another example of defining a character. The alien in the BUGZAP program on the Introductory Cassette was made up on the matrix in the drawing below.

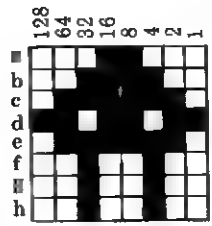


16 + 8 = 24
32 + 16 + 8 + 4 = 60
64 + 32 + 16 + 8 + 4 + 2 = 126
128 + 64 + 16 + 8 + 2 + 1 = 219
64 + 32 + 16 + 8 + 4 + 2 = 126
32 + 4 = 36
64 + 2 = 66
128 + 1 = 129

If you use the code 224 for the new character definition, here is the **VDU** statement which defines the complete character:

```
VDU 23,224,24,60,126,219,126,36,66,129 RETURN
```

By changing L7 and L8, the 'upright' alien shown in the drawing below can be defined. The code for this character must be a different one from the one above (eg 225), otherwise you will lose the original alien.



Have a go at defining this new character, then check the result by displaying it on the screen with

```
VDU 225 RETURN
```

The program below shows how you can produce an animated alien by using both these characters.

```
10 VDU 23,224,24,60,126,219,126,36,66,129
20 VDU 23,225,24,60,126,219,126,36,36,36
30 MODE 2
40 VDU 23,1,0;0;0;0
50 REPEAT
60 PRINT TAB(10,16);CHR$(224)
70 NOW% = TIME : REPEAT UNTIL TIME = NOW%+25
80 PRINT TAB(10,16); CHR$(225)
90 NOW% = TIME : REPEAT UNTIL TIME = NOW%+25
120 UNTIL FALSE
```

Line 40 gets rid of the flashing cursor, which would otherwise be a distraction. You can retrieve it by typing

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

Graphics

Introduction

The graphics instructions are pretty extensive in the Electron, and they all have certain things in common. The easiest commands to understand are the **MOVE** and **DRAW** commands, and these will be used for illustration in the following section. The ideas presented here are true for all graphics commands (including **CLG**).

REMEMBER: when you press **BREAK**, the computer is in mode 6. This is not a graphics mode and nothing will happen when you try to plot things. Always remember to go into a graphics mode to try these things out. Mode 1 is a good one to start with. Similarly, programs should always have a **MODE** command in them, as described at the beginning of this chapter.

The graphics coordinate system

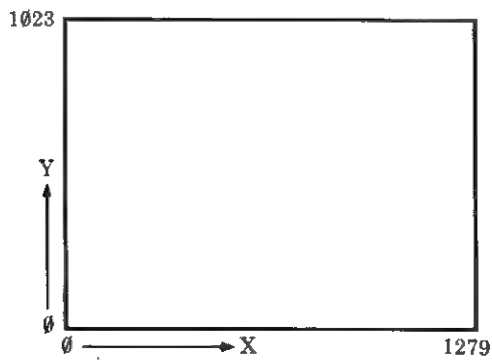
Firstly, we must describe the coordinate system, that is to say how positions of points are labelled. This is similar to the text coordinate system but there are three differences.

Firstly the system has the point **(0,0)** in the bottom left hand corner, and row numbers are labelled upwards.

Secondly, the top right hand point on the screen is **(1279,1023)**, the same in all modes (see the drawing below). This is so that drawing a line from, say **(100,100)** to **(400,400)** always draws a line in the same place, even though the pixel size varies with the mode.

Lastly, points off the screen are well defined, that is to say, drawing a line from, say **(–300,–400)** to **(300,400)** is perfectly legal, and what appears is what you would expect – that portion of the line that is in the area viewed by the screen.

The graphics screen



The graphics cursor is an invisible point on the screen, and is where you are about to draw from. Move it about the screen with the **MOVE** command, and drawing is easiest with the **DRAW** command. Thus

```
MOVE 100,100 RETURN
DRAW 400,400 RETURN
```

moves the cursor to **(100,100)** and draws a white line to **(400,400)**. Try lots of lines in different modes to get the feel of the coordinate system.

The GCOL command

Just as the foreground and background colours of text were defined using the **COLOUR** command, so the corresponding colours in graphics are defined using the **GCOL** command. Try the following:

```
MODE 1 RETURN
GCOL 0,1 RETURN
DRAW 300,300 RETURN
```

This draws a line in colour 1 (red) from **(0,0)** – where the graphics cursor is when the mode is changed – to **(300,300)**. However, you will notice that the **GCOL** command has *two* numbers after it. The second is just like the **COLOUR** command's number, that is the foreground colour number, or, if 128 is added to it, the background colour number. **CLG** is the graphics equivalent of **CLS** and clears the graphics area to the current graphics background colour. Thus

```
GCOL 0,129 RETURN
CLG RETURN
```

sets the graphics background colour to 1 and clears the graphics screen to this colour (red). Note that the **CLG** command is much slower than the **CLS** command.

The first number in the **GCOL** command is unusual. It tells the computer what to do with the graphics point. The following values are defined:

0 – write the point to the screen (what one would normally expect).
1 – OR the point to be plotted with what is on the screen.
2 – AND the point to be plotted with what is on the screen.
3 – EOR the point to be plotted with what is on the screen.
4 – INVERT what is on the screen, regardless of what colour is to be plotted.
5 – leave what is on the screen alone.

Other values do stripey things which may change with different releases of the software.

What is meant by OR, AND, EOR and INVERT? Each pixel has a colour – in mode 1 with four colours, this is from Ø to 3, or ØØ, Ø1, 1Ø and 11 in binary. What appears on the screen is the result of a logical operation between the pixel you want to plot and what is already on the screen. The OR and the AND are the same as for the BASIC commands. EOR means 'exclusive OR', which is the same as OR unless both bits are one, in which case the result is zero. Again, all this is most easily explained by specific examples. The following assumes that you are in mode 1.

Assume there is a red screen (from **GCOLØ,129 :CLG**, above). Setting **GCOL 1,2** sets the foreground colour to 2 (1Ø in binary), and the colour 'mode' to OR. Drawing a line then takes the red pixel on the screen (red – colour 1 = Ø1 in binary), and ORs it with the yellow pixels you are plotting. The pixel colour that appears is the Ø1 ORed with 1Ø, which is 11, colour 3, which is white. (Try it).

Given this white line on a red background, set the colour with **GCOL 2,2**, which has foreground colour 2 and colour mode AND. Plotting a line then takes what is on the screen and ANDs it with the yellow pixel, colour 2 or 1Ø in binary. Therefore when the line is plotting on the red background, Ø1 (red) is ANDed with 1Ø (yellow), then result being ØØ (black). If it crosses the white line – the white pixels (11) are ANDed with the yellow pixels (1Ø) to give 1Ø (yellow).

Setting **GCOL 3,131** sets the background colour to 3 (white) and the colour mode to EOR. Doing a **CLG** then EORs its pixel with 11 – that is to say ØØ goes to 11, Ø1 to 1Ø, 1Ø to Ø1 and 11 to ØØ. The screen is thus inverted in colour, and repeating the command restores it to its original state.

If this does not seem too clear, playing around with it for a little should help. It has two main purposes – setting the EOR mode allows erasure of a line by plotting it again. In four colour modes, two independent two-colour pictures may be drawn and selectively displayed using the palette.

The PLOT command

MOVE and **DRAW** are two special cases of the more general **PLOT** command, which is as follows.

```
PLOT K%,X%,Y%
```

where **K%** is the plot mode (ie what you are actually going to do); **X%** and **Y%** are the coordinates of the point to which you are plotting.

K% takes the following values:

Ø	Draw a line, relative (that is **X%** and **Y%** are displacements from the current graphics cursor position), with no change on the screen.
1	As Ø, but draw in foreground colour.
2	As Ø, but invert what is on the screen (colour mode 4 forced).
3	As Ø, but draw in background colour.
4 to 7	As Ø to 3 but plot absolute (plot to the point **X%,Y%**).
8 to 15	As Ø to 7, but plot the last point twice. This is so that when plotting in inverted modes, the line is continuous.
16 to 31	As Ø to 15 but with a dotted line.
32 to 63	Reserved for the graphics extension ROM.
64 to 71	As Ø to 7, but plot the specified point only.
72 to 79	Fill sideways on background colour (see below).
8Ø to 87	Plot triangles (see below).
88 to 95	Fill right on non-background colour (see below).
96 to 255	Reserved for graphics extension ROM.

Advanced graphics

Triangle plotting

This plots a solid triangle using as vertices the point specified, the graphics cursor and the previous graphics cursor. This can be used to fill many different shapes.

Sideways filling on background colour

This plots the line sideways from the specified point, left and right, until either the edge of the window is reached or the line meets a pixel of non-background colour. The graphics cursor is set to one of the end points, and the previous graphics cursor (used in triangle plotting) to the other.

The values of these may be found out via **FX** call number 13 (decimal). If the point specified is outside the graphics window, or is not on background colour, then the Y coordinates of the returned points are different.

Filling right

Filling right until background colour plots right from the specified point as far as either the edge of the graphics window or a pixel of background colour is found. The endpoints (note that it does not go left) are retrievable via **FX 13** in the same way. The endpoint is actually the first pixel of background colour found, thus if the specified point is background colour, the endpoint returned is the same as the specified one.

These two routines together enable a fast routine to be constructed to fill any enclosed shape.

The VDU command

The **VDU** command writes a series of bytes to the screen in a similar way to the **PRINT** command. Thus the following two commands are exactly the same:

```
VDU 12,65,66,67
```

```
PRINT CHR$(12); CHR$(65); CHR$(66); CHR$(67
);
```

(Note the semicolon at the end of the **PRINT** statement – the **VDU** command does not send a carriage return unless you explicitly tell it to). Most numbers that you need to write to the **VDU** are single bytes (characters, for example). However, the graphics coordinates are all double byte quantities and are sent lower byte first, higher byte second. The **VDU** command enables this to be done easily. If a number in the **VDU** command is followed by a semicolon, that number is interpreted as a double byte quantity. If you are unsure of bytes and double bytes, the quick rule is that if you are doing a graphics operation using the **VDU** command, you must always follow a graphics coordinate with a semicolon. This only applies to the **VDU** command.

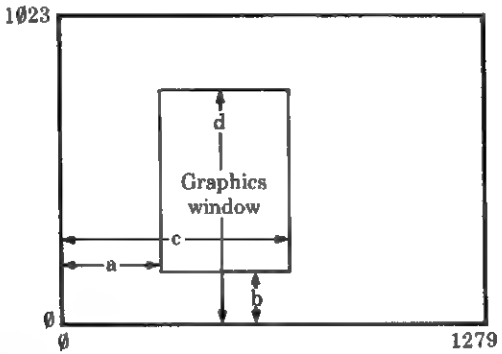
There are two more graphics commands, both of which are done via the **VDU** command.

Graphics windows

Just as text may have a text window defined, outside of which no text command has effect, so a graphics window may be similarly defined. This is done with

```
VDU 24,L%;B%;R%;T%;
```

where **L%,B%** and **R%,T%** are the coordinates of the lower left and upper right pixels inside the window. Setting a window thus prevents any plotting outside it. Also, because **CLG** is just another plot command, defining a graphics window and doing a **CLG** is a quick way of plotting rectangles.



The graphics origin

So far it has been said that the point **(0,0)** is at the bottom left hand corner of the screen. This point (called the origin) may be specified to live elsewhere with the origin command.

```
VDU 29, X%; Y%;
```

sets the position of the origin on the screen for future graphics commands. Thus to set the origin in the middle of the screen, use **VDU 29,640;512;**. It does not move the physical position of what is on the screen, the graphics windows or the graphics cursor.

Plotting characters

If **VDU 5** is entered, the text and graphics cursors are said to be joined, that is text appears at the graphics cursor which then moves as the text is written. This is mainly used for labelling graphs. The graphics cursor points to the top right pixel of the 8 by 8 character cell to be written, and is moved 8 pixels along by writing letters. This is seen in the following program.

```
10 MODE1
20 VDU5
30 REM All the text now appearing is 'plott
ed'
40 DRAW 500,500
50 PRINT "Hello mummy";
60 REM the last print statement moved the g
raphics cursor
70 REM   as can be seen by the next plot st
atement.
80 DRAW 0,0
90 END
```

VDU4 restores the text cursor.

The palette

Colours defined through the **COLOUR** and **GCOL** commands are more properly referred to as logical colours. When a mode is changed, these logical colours appear as certain physical colours, thus in mode 1, colour 1 is red and colour 2 is yellow. The palette allows this to be changed, thus colour 1 may be made to be blue and colour 2 flashing black on white. To be exact, we must distinguish between two types of colour:

The logical colour is what is output by the **COLOUR** commands. The maximum logical colour is limited by the number of colours available in the mode.

The physical colour is what appears on the screen. The physical colours and their numbers are listed below.

Physical number	Display colour
Ø	Black
1	Red
2	Green
3	Yellow
4	Blue
5	Magenta
6	Cyan
7	White
8	Flashing black/white
9	Flashing red/cyan
1Ø	Flashing green/magenta
11	Flashing yellow/blue
12	Flashing blue/yellow
13	Flashing magenta/green
14	Flashing cyan/red
15	Flashing white/black

Each logical colour has a physical colour assigned to it, which may be changed by reprogramming the palette. This is done as follows:

```
VDU 19,L%,P%,Ø,Ø,Ø
```

where **L%** is the logical colour and **P%** is the physical colour. So in mode 1, to change all the pixels with logical colour 3 (usually white) to blue (logical colour 4), the command **VDU 19,3,4,Ø,Ø,Ø** is used. Thus while the very detailed mode Ø is a two colour mode, the colours themselves may be anything available from the palette, such as green on red. Note also that the palette reprogramming is very fast as it does not involve a lot of the screen memory being reprogrammed.

21 VDU codes

Introduction

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.

The VDU drivers interpret all 32 ASCII control character codes. Many of the ASCII control codes are followed by a number of bytes. The number of bytes which follow depends on the function to be performed. The VDU code table summarises all the codes and gives the number of bytes which follow the ASCII control code.

Detailed description

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 **VDU5** has been given only one cursor is active (the graphics cursor). This enables text characters to be placed at any position on the screen. There are a number of other effects: text characters overwrite what is already on the screen so that characters can be superimposed; text and graphics can only be written in the graphics window and the colours used for both text and graphics are the graphics colours. In addition the page no longer scrolls up when at the bottom of the page. Note however that **POS** and **VPOS** still give you the position of the text cursor.

VDU code table

Decimal	Hex	CTRL	ASCII abbreviation	Bytes extra	Meaning
Ø	Ø	@	NUL	Ø	Does nothing
1	1	A	SOH	1	Reserved
2	2	B	STX	Ø	Reserved
3	3	C	ETX	Ø	Reserved
4	4	D	EOT	Ø	Write text at text cursor
5	5	E	ENQ	Ø	Write text at graphics cursor
6	6	F	ACK	Ø	Enable VDU drivers
7	7	G	BEL	Ø	Make a short beep
8	8	H	BS	Ø	Backspace cursor one character
9	9	I	HT	Ø	Forwardspace cursor one character
1Ø	A	J	LF	Ø	Move cursor down one line
11	B	K	VT	Ø	Move cursor up one line
12	C	L	FF	Ø	Clear text area
13	D	M	CR	Ø	Move cursor to start of current line
14	E	N	SO	Ø	Page mode on
15	F	O	SI	Ø	Page mode off
16	1Ø	P	DLE	Ø	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
2Ø	14	T	DC4	Ø	Restore default logical colours
21	15	U	NAK	Ø	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	**PLOT K,x,y**
26	1A	Z	SUB	Ø	Restore default windows
27	1B	[	ESC	Ø	Reserved
28	1C	\	FS	4	Define text window
29	1D	]	GS	4	Define graphics origin
3Ø	1E	^	RS	Ø	Home text cursor to top left
31	1F	_	US	2	Move text cursor to x,y
127	7F		DEL	Ø	Backspace and delete

6 **VDU 6** is a complementary code to **VDU21**. **VDU21** stops any further characters being printed on the screen and **VDU6** re-enables screen output. A typical use for this facility would be to prevent a pass-word appearing on the screen as it is being typed in.

7 This code,which can be entered in a program as **VDU7** or directly from the keyboard as **CTRL** G, causes the computer to make a short 'beep'.

8 This code (**VDU8** or **CTRL** H) moves the text cursor one space to the left. If the cursor was at the start of a line then it will be moved to the end of the previous line. It does not delete characters – unlike **VDU 127**.

9 This code (**VDU 9** or **CTRL** I) 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 (**VDU11** or **CTRL** K) moves the text cursor up one line. If the cursor is at the top of the screen then the whole display will move down a line.

12 This code clears the screen – or at least the text area of the screen. The screen is cleared to the text background colour which is normally black. The BASIC statement **CLS** has exactly the same effect as **VDU12** or **CTRL** L. This code also moves the text cursor to the top of the text window.

13 This code is produced by the **RETURN** key. However its effect on the screen display if issued as a **VDU13** or **PRINT CHR$(13);** is to move the text cursor to the left hand edge of the current text line (but within the current text window, of course).

14 This code makes the screen display wait at the bottom of each page. It is mainly used when listing long programs to prevent the listing going past so fast that it is impossible to read. The computer will wait until a **SHIFT** key is pressed before continuing. This mode is called 'paged mode'. Paged mode is turned on with the **CTRL** N and off with **CTRL** O.

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 **COLOUR** is used for an identical purpose. **VDU17** is followed by one number which determines the new colour. See the BASIC keyword **COLOUR** for more details.

18 This code allows the definition of the graphics foreground and background colours. It also specifies how the colour is to be placed on the screen. The colour can be plotted directly, ANDed, ORed or Exclusive-ORed with the colour already there, or the colour there can be inverted. In BASIC this is called **GCOL**.

The first byte specifies the mode of action as follows:

0 Plot the colour specified
1 OR the specified colour with that already there
2 AND the specified colour with that already there
3 Exclusive-OR the specified colour with that already there
4 Invert the colour already there

The second byte defines the logical colour to be used in future. If the byte is greater than 127 then it defines the graphics background colour (modulo the number of colours available). If the byte is less than 128 then it defines the graphics foreground colour (modulo the number of colours available).

19 This code is used to select the actual colour that is to be displayed for each logical colour. The statements **COLOUR** (and **GCOL**) are used to select the logical colour that is to be used for text (and graphics) in the immediate future. However the actual colour can be re-defined with **VDU 19**. For example

```
MODE 5
COLOUR1
```

will print all text in colour 1 which is red by default. However the addition of

```
VDU 19,1,4,0,0,0    or    VDU 19,1,4;0;
```

will set logical colour 1 to actual colour 4 (blue). The three zeros after the actual colour in the **VDU 19** statement are for future expansion.

In MODE 5 there are four colours (Ø,1,2 and 3). An attempt to set colour 4 will in fact set colour Ø 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.

2Ø This code **VDU2Ø** 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
Ø=black
1=white

Four colour modes
Ø=black
1=red
2=yellow
3=white

Sixteen colour modes
Ø=black
1=red
2=green
3=yellow
4=blue
5=magenta
6=cyan
7=white
8=flashing black/white
9=flashing red/cyan
1Ø=flashing green/magenta
11=flashing yellow/blue

12=flashing blue/yellow
13=flashing magenta/green
14=flashing cyan/red
15=flashing white/black

21 This code behaves in two different ways. If entered at the keyboard (as **CTRL** U) it can be used to delete the whole of the current line. It is used instead of pressing the **DELETE** key many times. If the code is generated from within a program by either **VDU21** or **PRINT CHR$ (21)**; it has the effect of stopping all further graphics or text output to the screen. The VDU is said to be disabled. It can be 'enabled' with **VDU6**.

22 This VDU code is used to change MODE. It is followed by one number which is the new mode. Thus **VDU22,6** is exactly equivalent to **MODE6** (except that it does not change **HIMEM**).

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, and this is explained in Appendix D.

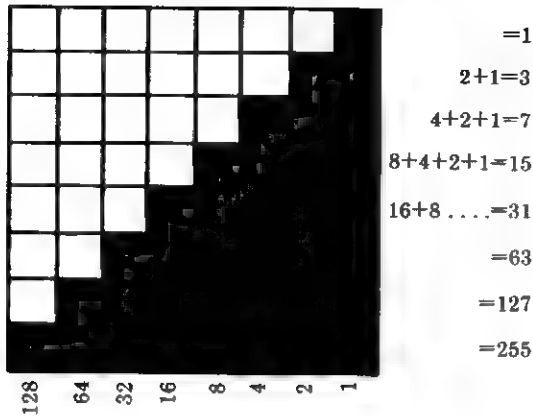
ASCII codes Ø to 31 are interpreted as VDU control codes – and this chapter is explaining the exact function of each. Thus the full ASCII set consists of all the VDU control codes, all the normal printable characters and a user defined set of characters.

For example if the user wishes to define ASCII code 24Ø 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
```

re-define character — 8 numbers giving the contents of each row of dots that makes up the desired character

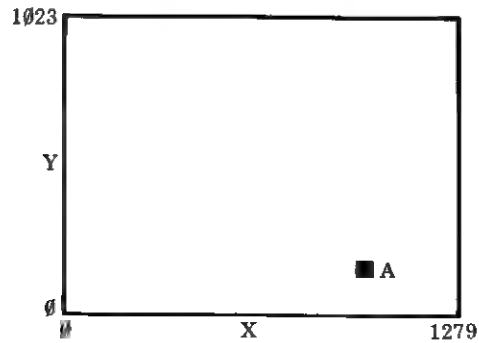


As explained above the user may define any ASCII code in the range 224 to 255. To display the resultant shape on the screen the user can type

```
PRINT CHR$(24Ø) or
VDU 24Ø
```

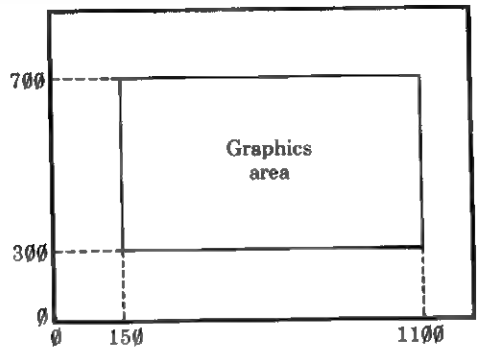
In the unlikely event of the user wishing to define more than the 32 characters mentioned above (ASCII 224 to 255) it will be necessary to allocate more RAM for the purpose.

24 This code enables the user to define the graphics window – that is, the area of the screen inside which graphics can be drawn with the **DRAW** and **PLOT** statements. The graphics screen is addressed with the following coordinates.



Thus the coordinates of A would be approximately 1ØØØ,2ØØ.

When defining a graphics window four coordinates must be given; the left, bottom, right and top edges of the graphics area. Suppose that we wish to confine all graphics to the area shown below.



The left hand edge of the graphics area has an X value of (about) 15Ø. The bottom of the area has a Y value of 3ØØ. The right hand side has X=11ØØ and the top has Y=7ØØ. 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 coordinates have to be sent to the VDU software as two bytes since the values may well be greater than 255. The semi-colon punctuation in the VDU statement sends the number as a two byte pair with low byte first followed by the high byte.

25 This VDU code is identical to the BASIC **PLOT** statement. Only those writing machine code graphics will need to use it. **VDU25** is followed by five bytes. The first gives the value of A referred to in the explanation of **PLOT** in the BASIC keywords chapter. The next two bytes give the X coordinate and the last two bytes give the Y coordinate. Refer to the entry for **VDU24** for an explanation of the semi-colon syntax used. Thus

```
VDU 25,4,100;500;
```

would move to absolute position 1ØØ,5ØØ.

The above is completely equivalent to

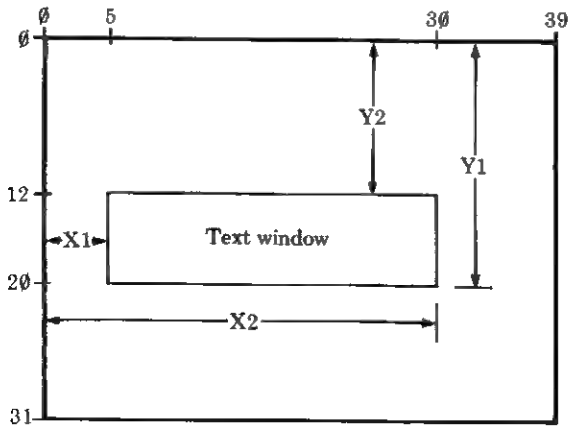
```
VDU 25,4,100,0,244,1
          X      Y
```

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.

28 This code (**VDU28**) is used to set a text window. Initially it is possible to write text anywhere on the screen but establishing a text window enables the user to restrict all future text to a specific area of the screen. The format of the statement is

```
VDU 28,leftX,bottomY,rightX,topY
```

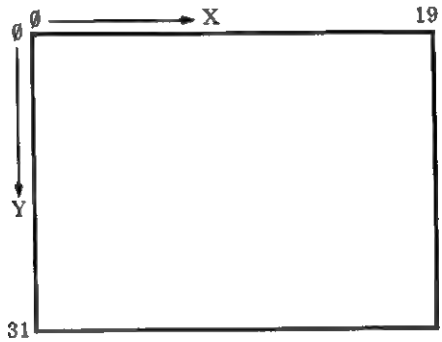
where **leftX** sets the left hand edge of the window
bottomY sets the bottom edge
rightX sets the right hand edge
topY sets the top edge



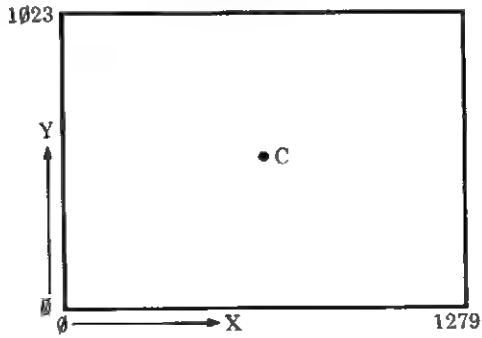
For the example shown the statement would be

```
VDU 28,5,20,30,12
```

Note that the units are character positions and the maximum values will depend on the mode in use. The example above refers to MODE1 and MODE4. In MODES 2 and 5 the maximum values would be 19 for X and 31 for Y since these modes have only 20 characters per line.



29 This code is used to move the graphics origin. The statement **VDU29** is followed by two numbers giving the X and Y coordinates of the new origin. The graphics screen is addressed



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 **VDU24** if you require an explanation of the trailing semi-colons. Note also that the graphics cursor is not affected by **VDU29**.

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

31 The code **VDU31** enables the text cursor to be moved to any character position on the screen. The statement **VDU31** is followed by two numbers which give the X and Y coordinates of the desired position.

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.

127 This code moves the text cursor back one character and deletes the character at that position. **VDU127** has exactly the same effect as the **DELETE** key.

128-223 These characters are normally undefined and will produce random shapes.

224-255 These characters may be defined by the user using the statement **VDU23.** It is thus possible to have 32 user defined shapes such as

```
♣ VDU 23,224,8,28,28,107,127,107,8,28
♦ VDU 23,225,8,28,62,127,62,28,8,0
♥ VDU 23,226,54,127,127,127,62,28,8,0
♠ VDU 23,227,8,28,62,127,127,127,28,62
```

Try typing each of the lines above, remembering to press the **RETURN** key after each definition. To display any of the new definitions, type in the appropriate VDU code. For example, to display the heart, type

VDU226 **RETURN**

Character definitions 224 to 255 are stored in a block of memory reserved for them in the computer. If however, you need more characters, or you want to re-define some of the keyboard characters, the best way to do this is to tell the computer to set aside extra memory to store them. (If you don't, you may run into problems). The operating system call ***FX2**▮ described in Appendix D enables you to do this.

22 Making sounds

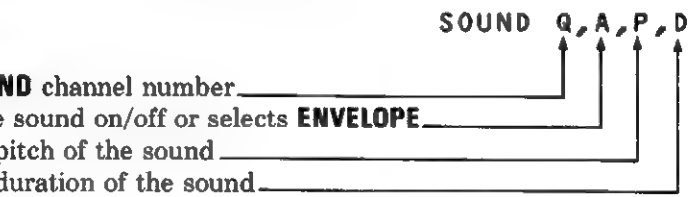
Introduction

Inside the Electron is a sound synthesiser which you can program to generate virtually any sound you like. The synthesiser is controlled by two BASIC commands: **SOUND** and **ENVELOPE**, and each command requires you to type in a series of numbers (or parameters) after it. These parameters determine the type of sound you will hear from the Electron's internal loudspeaker; ie the type of sound, pitch, duration, and so on.

There is a vast range of parameter values which gives you an almost unlimited range of possibilities when writing programs for the synthesiser. However, you don't need to be a wizard in order to use the sound system on a relatively simple level, and this chapter will serve to get you started. Once you've had some practice, read the last section in this chapter which goes into the **SOUND** commands in more detail.

The SOUND command

The **SOUND** command must be followed by four parameters which we will call **Q**, **A**, **P** and **D**. So the **SOUND** command takes the form



Press **ESCAPE** and type the following:

```
SOUND 1,-15,0,100 RETURN
```

This will produce a single tone of fairly long duration. Type this line in again and increase the P parameter (the third number). You'll notice that the pitch increases.

The Q parameter

There are four values of Q, and each one selects the **SOUND** channel number.

Q=Ø Channel Ø, selects noise
Q=1 Channel 1, selects tone
Q=2 Channel 2, selects tone
Q=3 Channel 3, selects tone

You may wonder why there are three **SOUND** channels which all select tone as opposed to noise. The only reason for this is to make the sound system as compatible with the BBC Microcomputer as possible. From now on, we will ignore channels 2 and 3.

The A parameter

This parameter does three things, depending on the value you give it:

When A is a negative number, the amplitude is at maximum, ie 'on'. When A is zero, the amplitude is at minimum, ie 'silence'. When A is between 1 and 16 inclusive, an **ENVELOPE** command of the same number is selected. Forget the **ENVELOPE** command for the moment, you will meet it later in this chapter.

Note that for the sake of compatibility with the BBC Microcomputer, use A=–15 for 'sound on', and A=Ø for 'silence'.

The P parameter

The value of this parameter controls the pitch of the generated sound. The range of values are from Ø to 255, and each consecutive value will give a quarter-semitone pitch change. The lowest note (P=Ø) is the B one octave and a semitone below middle C, and the highest accurate note is the B above middle C (P=1ØØ). Although values above P=1ØØ are not accurate to the even-tempered scale, they can still be put to good use for making sound effects. The table below is a quick reference guide to help you find the pitches you want.

	Octave number						
Note	1	2	3	4	5	6	
B	Ø	48	96	144	192	24Ø	
C	4	*52	1ØØ	148	196	244	*middle C
C#	8	56	1Ø4	152	2ØØ	248	
D	12	6Ø	1Ø8	156	2Ø4	252	
D#	16	64	112	16Ø	2Ø8		
E	2Ø	68	116	164	212		
F	24	72	12Ø	168	216		
F#	28	76	124	172	22Ø		
G	32	8Ø	128	176	224		
G#	36	84	132	18Ø	228		
A	4Ø	88	136	184	232		
A#	44	92	14Ø	188	236		

When the noise channel (channel Ø) is selected, the P parameter has the following effect:

P=Ø Tone – high pitch
P=1 Tone – intermediate pitch
P=2 Tone – low pitch
P=3 Tone – intermediate pitch
P=4 Noise – short period
P=5 Noise – intermediate period
P=6 Noise – long period
P=7 Noise – intermediate period

The D parameter
The value of this parameter sets the duration of the tone in steps of 5ØmS. So if D=1ØØ, then the duration will be (1ØØ × 5Ø)/1ØØØ, or five seconds. The maximum value of D is 254 (12.75 seconds).

If D=–1, then the tone will carry on until it is turned off.

Using the SOUND command in a program

Before using the **SOUND** command in a program, try following through the example below, in order to produce a single sound.

First of all, decide which channel to use: channel Ø will give noise, so use one of the three tone producing channels, say channel 1. So the first number is 1 – this is the value of Q.

Now choose the value for the second number – the A parameter. At the moment we aren't using any **ENVELOPE**s, and we want to hear the sound, so the next number is –15 – this is the value of A.

The third number – the P parameter – determines the pitch. If you look at the pitch table, the C below middle C is the value 4. So P=4.

How long shall the sound last? The fourth number – the P parameter – gives the duration in 5ØmS (five hundredths of a second). So for a duration of five seconds, P=1ØØ.

Now type the following **SOUND** command with the number above:

```
SOUND 1,-15,4,100 RETURN
```

This produces a tone which lasts for five seconds, and is the C below middle C.

Remembering that the pitch value is in quarter semitone steps, then to produce the sound corresponding to C (one semitone higher than the previous one), we must increase the pitch value by four.

```
SOUND 1,-15,8,100 RETURN
```

To play the next note up the scale, increase the P parameter by four again, and so on.

Instead of using numbers in the **SOUND** command, you can incorporate the **SOUND** command into a program, and use real or integer variables (see chapter 11). For example, let's produce a chromatic scale over an octave (equivalent to playing every note in sequence on a piano keyboard over an octave).

Instead of having to type in 13 **SOUND** commands to get the 13 different notes in the scale, we can use a variable for the P parameter, and change the variable value 13 times by using a **FOR . . . NEXT** loop. Try the program below which demonstrates this.

```
5 REM chromatic scale starting at C below m
iddle C up to middle C
10 FOR X%=4 TO 52 STEP 4
20 SOUND 1,-15,X%,5
30 NEXT
```

Lines 10 and 30 set up a loop with variable **X%**, whose values are passed to the pitch parameter in line 20. Line 20 contains the **SOUND** command: channel 1 is selected, the sound is 'on' (A=–15), the pitch is determined by the value of **X%**, and each sound plays for 5 × 50mS, or 0.25 seconds.

As you can hear, each new note only starts after the one before has finished. If you want to put a pause between each note, insert another **SOUND** command between lines 20 and 30, but turn it 'off' by giving the A parameter value 0. The length of the pause will of course be decided by the value you give the last parameter. Try this line:

```
25 SOUND 1,0,X%,10
```

(Obviously, the pitch value here can be anything you like). Now list the program.

```
5 REM chromatic scale starting at C below m
iddle C up to middle C
10 FOR X%=4 TO 52 STEP 4
20 SOUND 1,-15,X%,5
25 SOUND 1,0,X%,10
30 NEXT X%
```

When you **RUN** this program, you will hear a pause between each note. Try changing the values of **X%** and the duration of the tones and pauses to get different effects. If you want to make a crude sequencer, put the complete program into a **REPEAT . . . UNTIL FALSE** loop.

ENVELOPE

Each **SOUND** command, as we have seen, allows you to generate a single tone, whose pitch and duration is defined by you within the **SOUND** command. The **ENVELOPE** command allows you to change the single tone into something far more complex. Try typing the following:

```
10 SOUND 1,2,100,100
```

If you run this, you will hear a continuous tone which lasts for five seconds. Now add this line:

```
20 ENVELOPE 2,1,4,-4,4,10,20,10,0,0,0,0,0,0
```

When you run this program, you'll hear a sound rather like a police siren. Ignore all the numbers after the **ENVELOPE** command, except for the very first one, which defines the **ENVELOPE** number, in this case 2. The second parameter in the **SOUND** command selects **ENVELOPE** number 2; (**ENVELOPE** 2,...); the third parameter selects the starting pitch and the fourth parameter selects a duration of five seconds.

Note that once an **ENVELOPE** statement has been executed, it will stay in the computer until it is either re-defined, or until the computer is switched off. So even if you delete line 20 in the program above, when you run the program **ENVELOPE** number 2 will still be selected and used.

The ENVELOPE command

The **ENVELOPE** command is followed by 14 parameters, each separated by a comma. To make life easier, only the first eight actually do anything on the Electron, but the rest must be included for the sake of compatibility with the BBC Microcomputer. These eight parameters are shown below, and are followed by six zeros to make up the 14 parameters necessary.

```
ENVELOPE n,s,Pi1,Pi2,Pi3,Pr1,Pr2,Pr3,0,0,0,
0,0,0
```

If you want to run Electron programs containing **ENVELOPE** statements on the BBC Microcomputer, then you should get into the habit of always using the following numbers for the last six parameters:

```
ENVELOPE x,x,x,x,x,x,x,x,126,0,0,-126,126,1
26
```

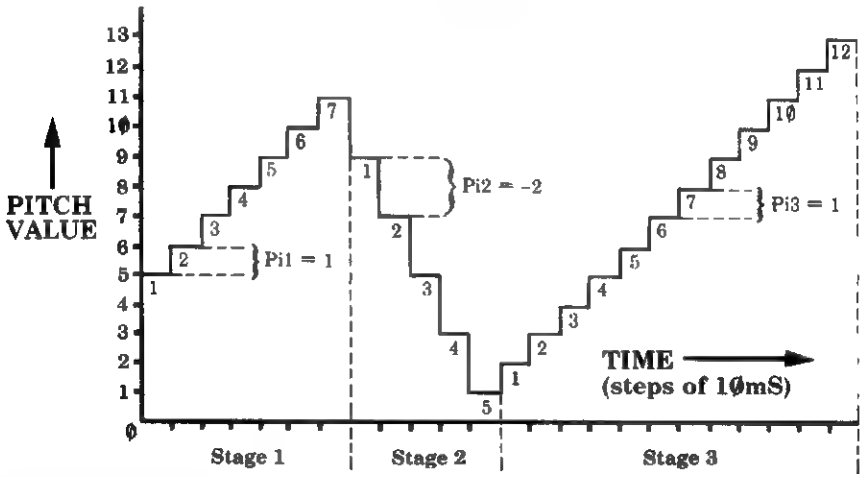
This will ensure that the resulting sound will be the same on both computers. For those of you who are not concerned with compatibility, use any numbers you like. In the rest of this section, 'zeros' have been used to save initially confusing you with too many numbers!

The duration of the **ENVELOPE** effect varies, depending on the values in the **ENVELOPE** command. However, as long as the duration is less than that of its associated **SOUND** command, it will repeat itself until the **SOUND** command finishes. For example, if you set the duration of a **SOUND** command to five seconds and select an **ENVELOPE** which makes a 'WOW' sound lasting for one second, then the result will be 'WOWOWOWOWOW'. With the **ENVELOPE** command, you can:

- Alter the **SOUND** pitch parameter by a specified increment
- Specify how many increments you want
- Set the length of time each increment 'plays'

The length of each increment can only be specified once in the **ENVELOPE** command, but the increment value itself, and the number of increments can both be specified three times!

Have a look at the graph below, which represents pitch plotted against time for an **ENVELOPE** command. As you can see, the resulting envelope contains three stages. In the first stage, the pitch increases by an increment of 1, then by –2 in the second stage, then 1 in the third stage. In the first stage there are seven pitch increments, in the second stage five, and in the third stage 12. (Notice that the duration of every pitch increment is the same).



The first eight parameters of the **ENVELOPE** command are as follows.

First parameter n
This is the **ENVELOPE** number, and is selected by the second parameter of the **SOUND** command. Values are from 1 to 16.

Second parameter s
This gives the duration of every pitch change in the **ENVELOPE** command, (ie all three stages), and each value corresponds to 10mS. Values are from 0 to 255. Values 1 to 127 give durations of 1 × 10mS to 127 × 10mS,

and when the **ENVELOPE** is finished, it repeats itself until the duration of the associated **SOUND** command has run out. Values 128 to 255 are equivalent to 1 to 127, except that at the end of the **ENVELOPE**, the final pitch is held until the associated **SOUND** command has run out.

Third parameter Pi1
This gives the value of every pitch increment during the first stage. Values are from –128 to 127.

Fourth parameter Pi2
This gives the value of every pitch increment during the second stage. Values are from –128 to 127.

Fifth parameter Pi3
This gives the value of every pitch increment during the third stage. Values are from –128 to 127.

Sixth parameter Pr1
This gives the number of pitch increments in the first stage. Values are from 1 to 255.

Seventh parameter Pr2
This gives the number of pitch increments in the second stage. Values are from 1 to 255.

Eighth parameter Pr3
This gives the number of pitch increments in the third stage. Values are from 1 to 255.

Parameters 9 to 14
These parameters must be put into the **ENVELOPE** command, but their values will have no effect on the effect produced by the Electron's **ENVELOPE** command. In order to keep the command compatible with the BBC Microcomputer, these values should be 126,Ø,Ø,–126,126,126.

Constructing an ENVELOPE

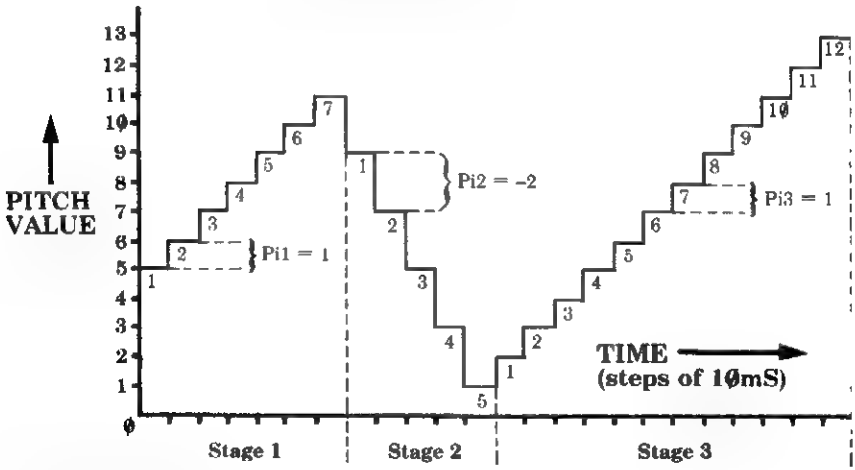
Type in the following program:

```
10 SOUND 1,2,4,50
20 ENVELOPE 2,1,1,-2,1,7,5,12,0,0,0,0,0,0
```

If you run this, you will hear a whirring sound. The sound is being transmitted on channel 1, **ENVELOPE** 2 is selected, the pitch value is 4, and the sound lasts for 5Ø × 5ØmS or 2.5 seconds.

The graph below shows the effect of the **ENVELOPE** on the sound. This graph is exactly the same as the one at the beginning of this section on the **ENVELOPE** command, and was drawn using the parameter values in the **ENVELOPE** statement above. The parameters are as follows:

n **ENVELOPE** number 2
s Pitch change duration is 1 × 1ØmS
Pi1 Stage 1 pitch changes INCREASE in increments of 1, (quarter semitones)
Pi2 Stage 2 pitch changes DECREASE in increments of 2, (half semitones)
Pi3 Stage 3 pitch changes INCREASE in increments of 1, (quarter semitones)
Pr1 Number of pitch changes in stage 1 is 7
Pr2 Number of pitch changes in stage 2 is 5
Pr3 Number of pitch changes in stage 3 is 12



The total time taken for this **ENVELOPE** command is 1ØmS × total number of pitch changes, or 24 × Ø.Ø1 seconds, = Ø.24 seconds. So the **ENVELOPE** is repeated over and over again, until the **SOUND** command finishes.

Additional SOUND features

The first parameter of the **SOUND** command can be given extra values which provide you with some more facilities. To use these values, you must enter the first parameter as a four digit hexadecimal number (which must be preceded by a & sign, to tell the computer that a hexadecimal number follows). The four digits in this number can be represented by HOFC, and here is a description of each digit.

H HOLD — H = Ø for compatibility with the BBC Microcomputer.

O — Not used, but should be entered with a value of Ø for compatibility.

F FLUSH — F = Ø sound is queued. This means that the sound won't start playing until the previous one has finished. This is only true if the previous sound is on the same channel as this one. If the channel numbers are different, then the latest **SOUND** instruction to be executed by the computer will immediately start playing in place of the previous one.

F = 1 sound is not queued. This means that as soon as the sound is executed by the computer, any sound previously playing will immediately stop and be replaced by this sound.

C CHANNEL — C = Ø noise channel
C = 1, 2 or 3 tone channels

Example SOUND and ENVELOPE programs

The program below turns the Electron into a keyboard instrument by using the 12345 and QWERTY keys.

```
10 REM Keyboard - Caps Lock on!
20 *FX12,4
30 *FX11,4
40 S$= " Q2W3ER5T6Y7UI900P"
50 REPEAT
60 SOUND &11,-15,INSTR(S$,GET$)*4,1
70 UNTIL FALSE
```

This next program takes a string of note names, and plays the notes. The range of notes you can play is from the C below middle C, to the B above middle C. To produce the full range in sequence enter the following: **cdefgabCDEFGAB (RETURN)**. This will give you two octaves in the key of C major.

```
10 REM Tune player
20 S$ = " c d ef g a bC D EF G A B"
30 REPEAT
40 INPUT "=>" T$
50 FOR N% = 1 TO LEN T$
60 P% = INSTR(S$,MID$(T$,N%,1))
70 SOUND 1,-15,P%*4,4
80 NEXT
90 UNTIL FALSE
```

Laser Zap! This program uses an **ENVELOPE** with a downwards pitch sweep. To fire the laser, press any key.

```
10 REM Zap!
20 ENVELOPE 1,129,-15,-8,-3,10,10,10,126,0,
0,-126,126,126
30 REPEAT
40 SOUND &11,1,255,5
50 UNTIL GET = FALSE
```

This last program uses a repeated **ENVELOPE** with an upwards pitch sweep to produce a spaceship take-off sound.

```
10 REM Liftoff
20 ENVELOPE 1,1,6,6,6,2,2,1,126,0,0,-126,12
6,126
30 FOR S% = 0 TO 220
40 SOUND 1,1,S%,1
50 NEXT
```

23 Address pointers, indirection operators

The Electron's memory

The computer's memory consists of 65536 locations (Ø to 65535), each containing 1 byte (8 binary digits). Half of the computer's memory can be written to or read from (called RAM); the other half can only be read from (called ROM).

Each location in memory has a unique address (like the address of your house) which is a four digit hexadecimal number. Location Ø in memory is &ØØØØ and location 65535 is &FFFF. The & sign means that the numbers which follow are in hexadecimal. The easiest way to look at the computer's memory is on a memory map. Overleaf is a simplified memory map for the Electron, showing on the right the address of each location.

Looking at the memory map, see how it is divided into the two types, RAM and ROM. All the programming which went into making the machine work is stored in the upper half of the memory, from &8ØØØ to &FFFF. Your BASIC programs are stored, unless the computer is told otherwise, starting at location &ØEØØ. This position where the program starts is assigned to a resident variable called **PAGE**. **PAGE** is an *address pointer;* it tells the computer at which address to start executing a program when you tell it to **RUN**. The location at which the BASIC program finishes is also assigned to a variable, called **TOP**. If you type:

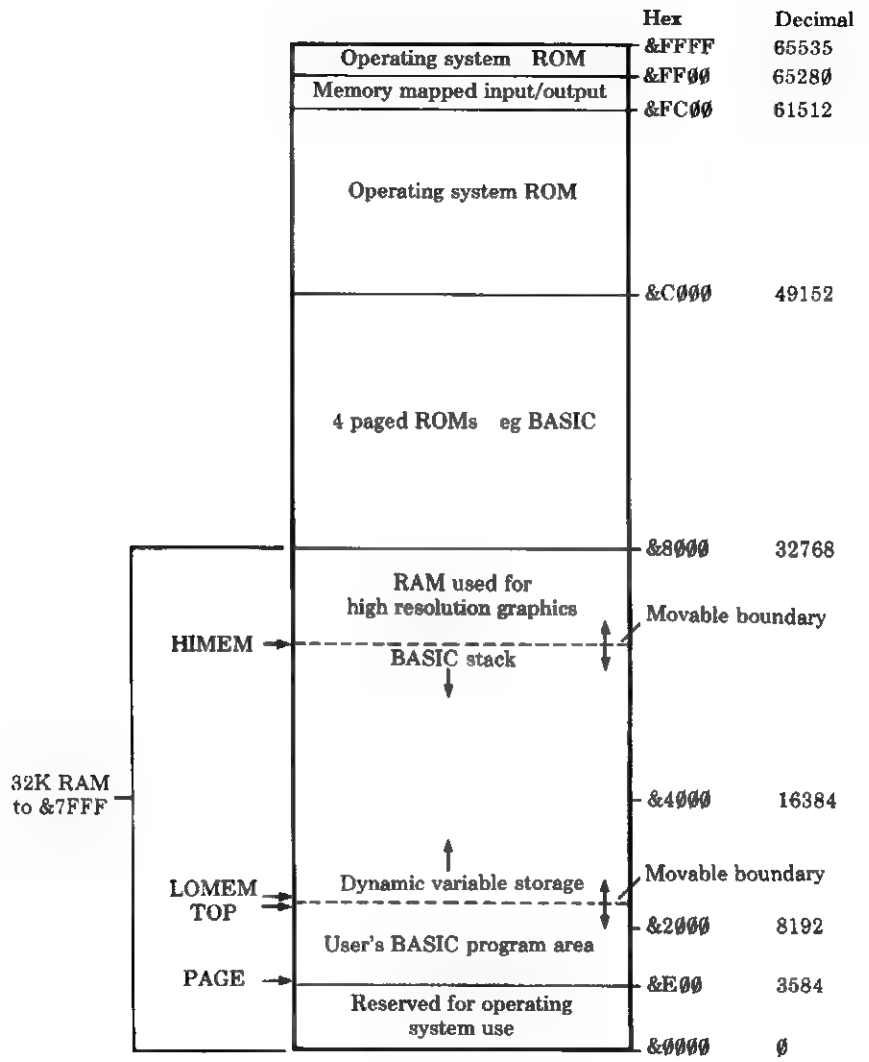
```
PRINT TOP-PAGE RETURN
```

the computer tells you how many bytes of memory your program fills.

Note that future expansions of the Electron, eg a disc filing system, will move **PAGE** up.

The next address pointer is **LOMEM**. This tells the computer where it can store the variables which are used by your program, and is usually the same value as **TOP**.

Memory map



The last address pointer is **HIMEM**. **HIMEM** shows the position of the bottom of the screen memory, so any program or variables must be kept below this value.

Programs are normally loaded at &ØEØØ, but they can be put higher up the memory by altering the value of **PAGE**. For example, if you make **PAGE** = &1ØØØ, and then you **LOAD** a program from tape, it will be situated at address &1ØØØ. When you do this, **TOP** and **LOMEM** are moved to their new position above the program. Another way in which to re-situate a program is to use ***LOAD**.

```
*LOAD "program name" 1000 RETURN
```

will **LOAD** the program from tape into the memory at location &1ØØØ. This instruction does not alter **PAGE**, and if you want to run the program you must make **PAGE** = &1ØØØ.

Any section of memory can also be saved by using ***SAVE**.

```
*SAVE "file name" SSSS FFFF EEEE RETURN
```

SSSS is the hex address from which you wish to start saving.

FFFF is the hex address plus 1 at which you wish to finish.

EEEE is the hex address at which execution should commence.

Indirection operators

Individual memory locations can be accessed from BASIC by using three indirection operators:

Symbol	Purpose	Number of bytes affected
?	Byte indirection operator	1
!	Double-word indirection operator	4
$	String indirection operator	1 to 256

To illustrate this, set a variable to an address in memory, for example:

```
A = &1000 RETURN
```

?A will give the contents of location A, so the contents of location &1ØØØ can be set by typing

```
?A = 100 RETURN
```

(Of course, because a location is a single byte, it cannot be set to a fractional number, or any integer above 255 decimal, which is &FF. If this is done, the least significant byte is stored in the memory location specified.)

To check the contents of &1000, type

```
PRINT ?A RETURN
```

BASIC integer variables, such as age%, are stored in four consecutive bytes of memory, and four bytes can be accessed using !.

```
!A = 70965 RETURN
```

Strings can be placed direct in memory, each character's ASCII code being stored in 1 byte of memory.

```
$A = "STRING"
```

The $ indirection operator appends a carriage return to the end of the string, so the above command would give 6 bytes of ASCII code for the word 'STRING', plus a byte containing &D which is the ASCII code for RETURN.

Notice that the indirection operator is **$A**, and not **A$** which is a BASIC string variable. The string in memory can, however, be assigned to a BASIC variable:

```
name$ = $A RETURN
```

Another way of using ? is with both a variable and a number.

A?6 gives the contents of location **A+6**, in this case location &1006.

To look at the contents of a group of memory locations, write a small program:

```
10 FOR I = 0 TO 15
20 PRINT "CONTENTS OF ";A+I;" ARE ";A?I
30 NEXT
```

The indirection operators described above are used and explained more thoroughly in the chapter on Assembly Language.

24 User-programmable keys

In the same way that **FUNC** and the alphabetic characters give BASIC keywords at a single stroke, you can program the keys marked 1 to Ø to give any string you choose.

For example,

```
*KEY1 "*CAT"
```

will cause **FUNC** 1 to print ***CAT** on the screen.

Control characters may be placed in the string, by typing the ¦ character. For example, **CTRL** M is ¦M, which performs the same function as the **RETURN** key. For a list of control characters, see Appendix A. So,

```
*KEY1 "*CAT ¦M"
```

will cause **FUNC** 1 to print ***CAT** on the screen, and to set the command in operation. Therefore, a whole program, or a series of commands can be stored in one key.

A useful routine is to have one key which returns the computer to MODE 6 and lists the program in paged mode:

```
*KEYØ "MODE 6 ¦M ¦N LIST ¦M"
```

Here is a key definition containing a small BASIC program:

```
*KEY3 "1Ø REPEAT ¦M 2Ø PRINT CHR$(RND(95)+3
1) ¦M  3Ø UNTIL VPOS=24 ¦M RUN ¦M"
```

For the more advanced

The **BREAK** key can be programmed also. It takes the value 1Ø. The following program cannot be stopped either by **ESCAPE** or **BREAK**:

```
1Ø ON ERROR GOTO 3Ø
2Ø *KEY1Ø "OLD ¦M RUN ¦M"
3Ø PRINT "YOU CAN'T STOP ME!"
4Ø REPEAT UNTIL FALSE
```

ON ERROR is described in chapter 27. Actually, this program can be halted by pressing **CTRL** **BREAK**.

CTRL **BREAK** is called a 'hard reset'. It resets everything very nearly the way it was when the machine was first switched on. When you try it you'll hear a beep and you'll see that the ♦ reappears after the message at the top of the screen.

BREAK on its own is called a 'soft reset'. It is roughly equivalent to pressing **ESCAPE** and entering the commands **NEW** and **MODE 6**.

The soft reset does not clear the ***KEY** definitions for example.

The five screen editing keys can also be re-defined, just like **BREAK**, after the issue of a ***FX** command.

Keys can also be loaded with the contents of BASIC variables. The instruction which does this is OSCLI, which stands for operating system command line interpreter. It can be used with any operating system call from BASIC (distinguishable by a preceding asterisk), for example, ***KEY**, ***SAVE**, ***LOAD**, and so on. Each BASIC variable assigned to the **KEY** definition must be converted into a string, and the asterisk omitted, as follows:

```
OSCLI "KEY" + STR$X + "LIST |M"
```

This will put **"LIST |M"** into **KEY** X, where X is a BASIC variable.

There is no limit to the number of BASIC variables which may be used in an OSCLI assignment, provided that they are all either string variables, or are turned into strings using **STR$**.

25 BASIC keywords

This chapter contains a description of every word in the Electron BASIC language. These words are called 'keywords'.

The syntax of each keyword is shown, and an explanation of the form used is given below.

{ }	denote possible repetition of the enclosed symbols, zero or more times
[ ]	enclose optional items
|	indicates alternatives from which only one should be chosen
<num-const>	means a numeric constant such as 4.7 or 112
<num-var>	means a numeric variable such as Y or width
<numeric>	means either a <num-const> or a <num-var>, or a combination of these in an expression, like 4*X+1
<string-const>	means a string enclosed in quotation marks like "JONCRAWFORD"
<string-var>	means a string variable, like A$ or NAME$
<string>	means either a <string-const> or a <string-var>, or an expression such as A$+"ELK"
<testable condition>	means something which is either **TRUE** or **FALSE**. Since both **TRUE** and **FALSE** have values, it is possible to use <numeric> instead of <testable condition>
<statement>	means any BASIC statement, like **PRINT** or **GOSUB** or **PROC**
<variable name>	means any sequence of letters or numbers which is an acceptable variable name

BASIC keywords

ABS

Absolute value

Abbreviation None FUNCTION

Description This function gives the modulus; that is, it strips the minus sign from the number variable or expression following it.

Examples **PRINT ABS(X)** will give 2 if X is –2
deviation = ABS(Temp1–Temp2)
root = SQR(ABS(Y))

Brackets are optional where sense is not affected.

Syntax <num-var> = **ABS** (<numeric>)

ACS

Arc-cosine

Abbreviation None FUNCTION

Description This function gives the angle, between **0** and **PI** in radians, whose cosine is the number variable or expression following **ACS**. This expression must be between –1 and 1 inclusive.

Examples **angle = ACS(0.5)**
course = ACS(–0.789)
ANGLE = ACS(AD/HY)

Brackets are optional where sense is not affected.

Syntax <num-var> = **ACS** (<numeric>)

ADVAL

SOUND channel buffer status

Abbreviation **AD.** FUNCTION

Description — Gives number of free spaces in **SOUND** buffers. **ADVAL (–5)** to **ADVAL (–8)** correspond to **SOUND** channels Ø to 3 respectively.

Examples
```
X=ADVAL(–7):PRINT"Free spaces in ch. 2=  ";X
IF ADVAL(–5)<>Ø THEN SOUND 2, . . .
```

Syntax — <num-var> = **ADVAL**(<numeric>)

AND

Logical AND

Abbreviation — **A.** — OPERATOR

Description — This is a logical operator which is most commonly used in an **IF . . . THEN** statement to combine two conditions and obtain a **TRUE** or **FALSE** result.

False **AND** False gives False
False **AND** True gives False
True **AND** False gives False
True **AND** True gives True

If this result is **TRUE** the computer will go on to the statement following the **THEN**. If the result is **FALSE** the computer will go on to the statement following the **ELSE**, but if the **ELSE** is absent it will go on to the next line.

Examples
```
IF X<5 AND X>Ø THEN PROCmiddle
IF Z=17 AND Y<7 THEN PRINT "YES" ELSE PRINT
"NO"
```

Comments — **AND** may also be used in the conditional part of a **REPEAT . . . UNTIL** loop.

Syntax — <num-var> = <numeric> **AND** <numeric>
<num-var> = <testable condition> **AND** <testable condition>

ASC

ASCII code

Abbreviation None FUNCTION

Description This function gives the ASCII character value of the first character in the string which follows it. If this string is null it gives –1.

Examples **PRINT ASC("JOHN")** will give 74 (see ASCII table)
IF ASC(A$) = 78 THEN NEXT.
X = ASC("m")*10

Brackets are optional where sense is not affected.

Syntax <num-var> = **ASC** (<string>)

ASN

Arc-sine

Abbreviation None FUNCTION

Description This function gives the angle, between **–PI/2** and **PI/2** in radians, whose sine is the number variable or expression following **ASN**. This expression must be between –1 and 1 inclusive.

Examples **PRINT ASN(OP/HY)**
angle = ASN(0.5)

Brackets are optional where sense is not affected.

Syntax <num-var> = **ASN** (<numeric>)

ATN

Arc-tangent

Abbreviation None FUNCTION

Description This function gives the angle, between **–PI/2** and **PI/2** in radians, whose tangent is the number variable or expression following **ATN**.

Examples **PRINT ATN(OP/AD)**
angle = ATN(–3)

Brackets are optional where sense is not affected.

Syntax <num-var> = **ATN** (<numeric>)

AUTO

Automatic line numbering

Abbreviation **AU.** or **FUNC** A COMMAND

Description This command gets the computer to print the next line number and a space each time you press **RETURN**.

The command has two optional parameters: the first is the starting line number, and the second is the interval between each subsequent line number. The default value of both these parameters is 10.

Examples **AUTO 100,5** will give line numbers 100, 105, 110, 115 etc.
AUTO by itself will give line numbers 10, 20, 30, 40, 50, etc.

Comments The largest allowable line number is 32767, and the largest allowable interval is 255.

You must press **ESCAPE** to get out of the **AUTO** mode.

Syntax **AUTO**[<num-const>[, <num-const>]]

BGET#

Read a byte from file

Abbreviation **B.#** FUNCTION

Description Reads a single byte from a previously opened file whose channel number follows (see chapter on file handling).

Examples **byte = BGET# channel**
character = BGET#A

Syntax <num-var> = **BGET#** <num-var>

BPUT#

Store a byte to file

Abbreviation **BP.#** STATEMENT

Description Stores a single byte on a previously opened file whose channel number follows (see chapter on file handling).

Examples
```
BPUT# channel, number
BPUT# file, Z MOD 256
```

Syntax **BPUT#** <num-var>, <numeric>

CALL

Call assembled machine-code subroutine

Abbreviation **CA.** STATEMENT

Description Used from BASIC to call a previously assembled machine-code subroutine. Similar in operation to a **PROC**, being capable of passing parameters. Used in preference to a **PROC** where long calculation is involved, and speed is at a premium.

Examples
```
30 CALL &2000

70 fraction = &16A5
80 CALL fraction

150 CALL fraction, string$, number, integer%, ?byte
```

Syntax **CALL** <numeric>{,<num-var>|<string-var>}

CHAIN

Load and run a program

Abbreviation **CH.** or **FUNC** K STATEMENT

Description An instruction which **LOAD**s and **RUN**s the program whose title is in the quotes. If the title is omitted the next program on the tape will be loaded. Can be used in one program to load another. NB all variables except the resident integer variables are cleared.

Examples **CHAIN "PROG1"**
CHAIN ""

Syntax **CHAIN** <string>

CHR$

Character code

Abbreviation **CHR.** FUNCTION

Description Gives the character whose ASCII code is the number variable or expression following **CHR$**.

Examples **PRINT CHR$(32)**
A$ = A$ + CHR$(code%)

Brackets are optional where sense is not affected.

Syntax <string-var> = **CHR$** (<numeric>)

CLEAR

Clear memory

Abbreviation **CL.** STATEMENT

Description This instruction takes away all the variable names in use, except the resident integer variables A% to Z%, and @%.

Examples **IF FNcrash > 30 THEN CLEAR**

Syntax **CLEAR**

CLG

Clear graphics screen

Abbreviation None STATEMENT

Description — Fills the graphics screen with current graphics background colour (which can be altered by the **GCOL** instruction). The graphics cursor is 'homed' to Ø,Ø bottom left of graphics screen.

Examples
```
IF X THEN CLG
```

Comments — **CTRL** P has same effect.

Syntax — **CLG**

CLOSE#

Close a file

Abbreviation — **CLO.#** — STATEMENT

Description — Tells the computer you have completely finished with the file whose channel number follows (see chapter on file handling).

Examples
```
CLOSE# (Channel)
CLOSE# file1
```

Comments — **CLOSE#Ø** closes all files.

Syntax — **CLOSE#** <numeric>

CLS

Clear text screen

Abbreviation — None — STATEMENT

Description — Fills text screen with current text background colour (which can be altered by the **COLOUR** instruction). The text cursor is homed to Ø,Ø the top left of the text screen.

Examples
```
IF X THEN CLS
```

Comments — **CTRL** L has same effect.

Syntax — **CLS**

COLOUR

Abbreviation **C.** or **FUNC** C STATEMENT

Description Used to select text screen foreground and background colour.

Standard colours, with their logical values, in each mode are as follows:

Foreground colour		Background colour	
Logical no	**Actual colour**	**Logical no**	**Actual colour**
Modes 0,3,4,6			
0	Black (0)	128	Black (0)
1	White (7)	129	White (7)
Modes 1,5			
0	Black (0)	128	Black (0)
1	Red (1)	129	Red (1)
2	Yellow (3)	130	Yellow (3)
3	White (7)	131	White (7)
Mode 2			
0	Black (0)	128	Black (0)
1	Red (1)	129	Red (1)
2	Green (2)	130	Green (2)
3	Yellow (3)	131	Yellow (3)
4	Blue (4)	132	Blue (4)
5	Magenta (5)	133	Magenta (5)
6	Cyan (6)	134	Cyan (6)
7	White (7)	135	White (7)
8 Flashing	black/white (8)	136 Flashing	black/white (8)
9 Flashing	red/cyan (9)	137 Flashing	red/cyan (9)
10 Flashing	green/magenta (10)	138 Flashing	green/magenta (10)
11 Flashing	yellow/blue (11)	139 Flashing	yellow/blue (11)
12 Flashing	blue/yellow (12)	140 Flashing	blue/yellow (12)
13 Flashing	magenta/green (13)	141 Flashing	magenta/green (13)
14 Flashing	cyan/red (14)	142 Flashing	cyan/red (14)
15 Flashing	white/black (15)	143 Flashing	white/black (15)

COLOUR takes one parameter, which is the logical value of the particular colour required, as given in the tables.

Examples

```
COLOUR 2
COLOUR 131
```

Comments — Colours used in each mode may be changed using **VDU19;**.

See chapter 2Ø

Syntax — **COLOUR** <numeric>

COS

Cosine

Abbreviation — None — FUNCTION

Description — This function gives the cosine of an angle, which must be in radians.

Examples

```
PRINT COS(3.142)
X = COS(y)
```

Brackets are optional where sense is not affected.

Syntax — <num-var> = **COS** (<numeric>)

COUNT

Count characters

Abbreviation — **COU.** — FUNCTION

Description — Counts the number of characters printed using **PRINT** since last carriage return.

Examples

```
1Ø PRINT "Happy Birthday "; COUNT
2Ø PRINT "Happy" ' "Birthday "; COUNT
>RUN
Happy Birthday 15
Happy
Birthday 9
```

Comments — Different from **POS**, which gives the position of the cursor from the left hand margin.

Syntax — <num-var> = **COUNT**

DATA

Data in program

Abbreviation **D.** STATEMENT

Description This enables you to store information in a program and to recall it using a **READ** instruction. The information can be string or numeric. (See chapter on **READ** and **DATA**).

Examples
```
10 READ A,B$,century
20 DATA 3,GEORGE,18
```

Syntax **DATA** <str-const>|<num-const>|<num-var>{, <str-const>|<num-const>|<num-var>}

DEF

Define function or procedure

Abbreviation None STATEMENT

Description Informs the computer that an **FN** or **PROC** is about to be defined. (See chapters on procedures and functions.)

Examples
```
10 DEF FNdouble(X) = X*2
```
```
10 DEF PROCdouble
20X = X*2
30 ENDPROC
```

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

DEG

Degrees

Abbreviation **FUNC** H FUNCTION

Description Converts radians into degrees.

Examples
```
angle = DEG(PI/6)
angle = DEG(ACS(0.78))
```

Brackets are optional where sense is not affected.

Syntax <num-var> = **DEG**<numeric>

DELETE

Delete program lines

Abbreviation **DEL.** COMMAND

Description This command will delete a section of program from the first line number stated to the second inclusive. Cannot be used in a program.

Examples **DELETE 100,150**

Comments To delete a single line, just type the line number and press **RETURN**.

Syntax **DELETE** <num-const>,<num-const>

DIM

Dimension of an array

Abbreviation None STATEMENT

Description Informs the computer of how much memory to reserve for a named array. (See chapter on arrays.)

Examples **DIM Date$(12,31)**
DIM X(100)

Comments **DIM** is also used to allocate space for machine-code programs.

Syntax **DIM** <num-var>|<str-var>
(<numeric>{,<numeric>})
DIM <num-var> <numeric>

DIV

Integer division

Abbreviation: None — OPERATOR

Description: This tells the computer to divide one number into another using integer arithmetic; this means the result will always be a whole number.

Examples: **17 DIV 2** gives 8, ie the number of times that 2 can be subtracted from 17 with a positive or zero remainder.

Comments: If numbers or variables are used which are not integers, then they will be truncated before the division is carried out.

8.1 DIV 2.9 gives 4.

Syntax: <num-var> = <numeric> **DIV** <numeric>

DRAW

Draw line on screen

Abbreviation: **DR.** or **FUNC** D — STATEMENT

Description: Will draw a line from the previous coordinates of the graphics cursor to the new ones given, in all graphics modes (Ø, 1, 2, 4 and 5). To move the graphics cursor use the **MOVE** instruction.

The screen is always Ø to 1279 on the X axis and Ø to 1Ø24 on the Y axis, regardless of which graphics mode you are in. The line is drawn in the current graphics foreground colour which can be changed by using the **GCOL** instruction.

Examples:

```
1Ø MODE 4
2Ø MOVE Ø, 512
3Ø DRAW 1279, 512
```

will draw a horizontal line half way up the screen.

See chapter 2Ø

Syntax: **DRAW** <numeric>, <numeric>

ELSE

Abbreviation **EL.** or **FUNC** E (See IF)

Description Used to provide an alternative course of action if the result of an **IF** statement is false.

Examples
```
IF A=0 THEN PRINT "YES" ELSE PRINT "NO"
IF B THEN 100 ELSE 200
```

Syntax **IF** <testable condition> **THEN** <statement> **ELSE** <statement>

END

Abbreviation None STATEMENT

Description Can be used to halt execution of a program. Its other use is to reset **TOP** after a **PAGE** move.

Examples
```
PAGE=&1600 : END
60 IF finished THEN END
```

Syntax **END**

ENDPROC

End of procedure

Abbreviation **E.** STATEMENT

Description This statement must conclude a **DEF PROC** as it tells the computer you have finished defining the procedure.

Examples
```
100 DEF PROCname
110 REM statement
120 REM statement
130 ENDPROC
```

Syntax **ENDPROC**

ENVELOPE

Abbreviation **ENV.** STATEMENT

Description This statement is used in conjunction with the **SOUND** statement to control the pitch of a sound. The **ENVELOPE** statement is followed by 14 parameters.

ENVELOPE n,s,Pi1,Pi2,Pi3,Pr1,Pr2,Pr3,126,Ø,Ø,−126,126,126

Parameter	Range	Effect
n	1 to 4	Envelope number
bits Ø–6	Ø to 127	Length of each step in 1/1ØØ of a second
s bit 7	Ø or 1	Ø = auto-repeat the envelope 1 = no auto-repeat
Pi1	−128 to 127	Change in pitch per step in section 1
Pi2	−128 to 127	Change in pitch per step in section 2
Pi3	−128 to 127	Change in pitch per step in section 3
Pr1	Ø to 255	Number of steps in section 1
Pr2	Ø to 255	Number of steps in section 2
Pr3	Ø to 255	Number of steps in section 3

See chapter 22

Syntax **ENVELOPE** <num-var>,<num-var>,<num-var>,<num-var>,<num-var>,<num-var>,<num-var>,<num-var>,<num-var>,<num-var>,<num-var>,<num-var>,<num-var>,<num-var>

EOF#

End of file check

Abbreviation None — FUNCTION

Description This function is used to discover whether the end of an open file has been reached. The function gives a −1 if the end has been reached and a 0 if not. **EOF#** must be followed by the channel number.

Examples
```
IF EOF#(channel) THEN PROCclose
REPEAT...:UNTIL EOF#(X).
```

Syntax <num-var> = **EOF#**(<num-var>)

EOR

Logical exclusive-OR

Abbreviation None — OPERATOR

Description This is used in an **IF...THEN** or **REPEAT...UNTIL** loop to combine two conditions in the following way:

False **EOR** False gives False
False **EOR** True gives True
True **EOR** False gives True
True **EOR** True gives False

In other words, if the results of the two conditions combined by an **EOR** are different then the result is true.

Examples
```
IF A=6 EOR B<10 THEN GOSUB 420
```

Syntax <num-var> = <numeric> **EOR** <numeric>

EQUB

Abbreviation None STATEMENT

Description Used to insert a byte of data into an Assembly Language program. **EQUB** can only be used inside the square brackets enclosing a piece of Assembly Language.

Examples
```
EQUB 13
EQUB A%
```

Syntax **EQUB** <numeric>

EQUD

Abbreviation None STATEMENT

Description Used to insert a double-word (4 bytes) of data into an Assembly Language program. **EQUD** can only be used inside the square brackets enclosing a piece of Assembly Language.

Examples
```
EQUD 10000000
EQUD F%
```

Syntax **EQUD** <numeric>

EQUS

Abbreviation None STATEMENT

Description Used to insert the ASCII values of a string into an Assembly Language program. **EQUS** can only be used inside the square brackets enclosing a piece of Assembly Language.

Examples **EQUS "Too big"**
EQUS L$

Comments Used, among other things, for printing error messages in Assembly Language programs. Unlike the indirection operator **$**, **EQUS** does not add a **RETURN** (&D) to the end of the string.

Syntax **EQUS** <string>

EQUW

Abbreviation None STATEMENT

Description Used to insert a word of data (2 bytes) into an Assembly Language program. **EQUW** can only be used inside the square brackets enclosing a piece of Assembly Language.

Examples **EQUW &FFE0**
EQUW Z%

Syntax **EQUW** <numeric>

ERL

Error line number

Abbreviation None FUNCTION

Description A function which gives the line number in which the last error occurred.

Examples **X = ERL**
REPORT: PRINT " at line ";ERL

Syntax <num-var> = **ERL**

ERR

Error

Abbreviation None FUNCTION

Description A function which gives the numeric code for the last error which occurred. This is useful for error trapping.

Examples **IF ERR = 17 THEN CLOSE# (channel)**

Syntax <num-var> = **ERR**

EVAL

Evaluate

Abbreviation **EV.** FUNCTION

Description Mainly used to enable you to type an expression, such as a mathematical equation, into the computer while a program is running. The equation is entered as a string, eg **A$ = "COS(X/2Ø)"**, and **EVAL(A$)** will work it out.

Examples
```
A$ = "COS(X/2Ø)"
Y = EVAL(A$)
```

Syntax
<num-var> = **EVAL** (<string>)
<str-var> = **EVAL** (<string>)

EXP

Exponent

Abbreviation None FUNCTION

Description This mathematical function calculates the exponential e (2.7183 . . .) raised to any specified power.

Examples **Y = EXP(X)**
ie Y = e to the power of X

Syntax <num-var> = **EXP** (<numeric>)

EXT#

Reserved for future use.

FALSE

Abbreviation **FA.** CONSTANT

Description This is a condition which the computer understands to be the number Ø. If the computer decides a certain condition is false it will represent it as Ø, and will act accordingly.

Examples
```
REPEAT...:UNTIL FALSE
IF A = FALSE THEN...
```

Comments
```
PRINT 1 = 2
```
gives Ø, because 1 is not equal to 2, and so 1 = 2 is **FALSE**.

Syntax <num-var> = **FALSE**

FN

User-definable function

Abbreviation None Prefix

Description **FN** is a prefix that identifies a function, both in a **DEF** statement and in a function call.

Examples
```
1Ø INPUT A
2Ø answer = FNsquare(A)
3Ø PRINT answer
4Ø END
5Ø DEF FNsquare(number) = number*number

DEF FNe = 2.7182818.
```

Syntax **DEF FN** <name> [(<num-var>|<str-var>{,<num-var>|<str-var>})]

FOR

Abbreviation **F.** STATEMENT

Description **FOR** is used to initiate the control variable of the **FOR... NEXT** loop and will always take the following format:

FOR			**TO**	**STEP**
(numeric variable)	=	(number or numeric variable)	(n or nv)	(n or nv)
control variable		start parameter	finish parameter	increment

If the **STEP** is omitted it is assumed to be 1.

When executing a **FOR... NEXT** loop the computer sets the control variable to the start parameter.

Each time **NEXT** is encountered the computer increments the control variable and loops back to the instruction just after **FOR**. This is repeated until the control variable is greater than the finish parameter. NB the increment can be negative.

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

GCOL

Graphics colour

Abbreviation **GC.** STATEMENT

Description Used to select graphics screen foreground and background colour, and to control effect of mixing colours. Takes two parameters, the second being the logical value of the colour required, the first the way in which two colours mix.

Action of first parameter is as follows:

Ø Plot the colour specified
1 OR the colour specified with those already on the screen

2 AND the colour specified with those already on the screen
3 EOR the colour specified with those already on the screen
4 Plot the logical inverse of the colour specified

This mixing is carried out on a bit by bit basis. For 1, 2, and 3, each binary digit in the plotted colour's logical value is ORed, ANDed, or EORed with its respective digit in the screen colour's logical value, to produce the logical colour which is to be plotted on that part of the screen.

Inversion, 4, only involves the plotted colour, all its binary digits being inverted. In bit by bit logic, Ø is false and 1 is true.

The truth tables for OR, AND, and EOR are as follows:

Ø OR Ø gives Ø
Ø OR 1 gives 1
1 OR Ø gives 1
1 OR 1 gives 1

Ø AND 1 gives Ø
Ø AND Ø gives Ø
1 AND Ø gives Ø
1 AND 1 gives 1

Ø EOR Ø gives Ø
Ø EOR 1 gives 1
1 EOR Ø gives 1
1 EOR 1 gives Ø

Examples

```
GCOL 2, 1
GCOL RND(5)-1, RND(8)-1
GCOL, mix%, 129
```

Comments

Colours used in each mode may be changed using **VDU19;**

See chapter 2Ø

Syntax

GCOL <numeric>,<numeric>

GET

Get code from keyboard

Abbreviation None FUNCTION

Description This instruction causes the computer to read a character from the keyboard buffer. If there is none, the computer will wait for a key to be pressed. It then gives the ASCII code for that key (see ASCII table) before continuing.

Examples **Key = GET**

Comments The keyboard buffer may be flushed by ***FX15**.

Syntax <num-var> = **GET**

GET$

Get character from keyboard

Abbreviation **GE.** FUNCTION

Description This instruction is the same as **GET**, but gives a string containing the character before continuing.

Examples **Key$ = GET$**

Syntax <string-var> = **GET$**

GOSUB

Go to a subroutine

Abbreviation **GOS.** STATEMENT

Description This instruction tells the computer to go to a subroutine and start executing instructions from the specified line number until the instruction **RETURN**, when the computer must return to the instruction immediately after the **GOSUB** call. No more than 26 nested subroutines are allowed.

Examples
```
GOSUB 1000
ON A GOSUB 10, 20, 30
```

Comments It is possible to use an expression, and brackets must then be used:

```
GOSUB (10*A)
```

but this will not work if the program is **RENUMBER**ed.

Syntax **GOSUB** <numeric>

GOTO

Go to a line number

Abbreviation **G.** or **FUNC** G STATEMENT

Description This instruction tells the computer to jump to the specified line number and start executing instructions there.

Examples
```
GOTO 100
ON A GOTO 10, 20, 30
```

Comments It is possible to use an expression:

```
GOTO (10*A)
```

but this will not work if the program is **RENUMBER**ed.

Syntax **GOTO** <numeric>

HIMEM

Abbreviation **H.** VARIABLE

Description Address pointer containing the address of the lowest location in memory used by the screen display. Its value may change depending upon which mode you are

using. No BASIC instructions or variables are stored above this point. **HIMEM** can be altered by the user, to preserve space for machine-code programs between BASIC and the screen. **HIMEM** contains the highest memory location that BASIC may use. This may be changed after a **MODE** statement.

Examples
```
PRINT HIMEM
HIMEM = &4000
```

Syntax <num-var> = **HIMEM**

IF

Conditional IF

Abbreviation None

STATEMENT

Description A word forming part of the **IF . . . THEN . . . ELSE** statement. **IF** must always be followed by a testable condition, and the result of this test (**TRUE** or **FALSE**) will control the subsequent action by the computer.

Examples
```
IF A<5 THEN...
IF IQ>140 THEN PRINT "GENIUS"
```

Comments
```
IF A THEN 110
```
means
```
IF A<>0 THEN GOTO 110
```

Syntax **IF** <testable condition> **THEN** <statement>
IF <testable condition> **THEN** <line number>
[**ELSE** <statement> | <line number>]

INKEY

Abbreviation None

FUNCTION

Description (i) This instruction will wait a specified length of time for a key to be pressed before continuing. If a key is pressed its ASCII code is given but if no key is pressed a −1 is given. The number in brackets is the time delay

measured in 1/1ØØ sec, and can have any value between Ø and 32767.

Examples

```
key = INKEY(1ØØ)
```

Comments

In fact, this instruction and **GET**, **GET$**, and **INKEY$** will actually test the keyboard buffer. This means that an **INKEY** instruction will respond to any previously pressed key whose code has gone into this buffer (memory), even if you are not pressing it at this moment.

To get over this problem the keyboard buffer can be flushed using a ***FX15,1** instruction just before using **INKEY**. Also, the autorepeat can be turned off by using ***FX11,Ø**.

Description (ii)

In addition to the above, the **INKEY** instruction can be used to test for a single key directly. Using a negative number in the brackets, one for each key according to the table shown below, **INKEY** gives −1 if the key is pressed, Ø if it is not. **INKEY** used in this way does not read the buffer – it reads the key itself. See the table which follows.

Examples

```
IF INKEY(-99) THEN...
```

will be **TRUE** when the space-bar is pressed.

Brackets are optional where sense is not affected.

Key	Number	Key	Number
A	−66	1	−49
B	−1Ø1	2	−5Ø
C	−83	3	−18
D	−51	4	−19
E	−35	5	−2Ø
F	−68	6	−53
G	−84	7	−37
H	−85	8	−22
I	−38	9	−39
J	−7Ø	Ø	−4Ø
K	−71	−	−24
L	−87	;	−88
M	−1Ø2	:	−73
N	−86	,	−1Ø3
O	−55	.	−1Ø4
P	−56	/	−1Ø5
Q	−17	SPACE BAR	−99
R	−52	ESCAPE	−113
S	−82	CAPS LK	−65
T	−36	CTRL	−2
U	−54	SHIFT	−1
V	−1ØØ	DELETE	−9Ø
W	−34	COPY	−1Ø6
X	−67	RETURN	−74
Y	−69	↑	−58
Z	−98	↓	−42
		←	−26
		→	−122

Syntax <num-var> = **INKEY**(<numeric>)

INKEY$

Abbreviation **INK.** FUNCTION

Description The same as **INKEY** but gives the key as a character.

Examples **Key$ = INKEY$(1ØØ)**

Brackets are optional where sense is not affected.

Syntax <string-var> = **INKEY**(<numeric>)

INPUT

Abbreviation **I.** or **FUNC** I STATEMENT

Description This instruction halts the program, prints a ? on the screen, and waits for some information to be entered followed by **RETURN**.

INPUT must be followed by a variable.

Examples
```
INPUT X
INPUT name$, number
```

Syntax Too complex for a simple description.

INPUT# Input from file

Abbreviation **I.#** STATEMENT

Description **INPUT#** is like **INPUT**, but instead of receiving information from the keyboard the computer takes it from a previously opened file. Must be followed by channel number (see chapter on file handling).

Examples
```
INPUT# channel, make$, price
INPUT# C,B,A,Z$
```

Syntax
```
INPUT# <num-var>,<num-var>|<string-var>
{,<num-var>|<string-var>}
```

INSTR

In string

Abbreviation **INS**. Includes **(** FUNCTION

Description This function will give the position of one string within another, the leftmost character position being 1. The search normally starts from the beginning of the string but an optional third parameter provides the facility to start the search from any specific character position. The number given by **INSTR** is the position of the second string within the first. A search for a null string ("") will always give 1. If the search fails (the two strings are not the same at any position) then Ø is given.

Examples
```
position = INSTR(first$, second$, start)
PRINT INSTR("MONOTONOUS","ON")
```

will print 2, whereas

```
PRINT INSTR("MONOTONOUS","ON",3)
```

will print 6

Syntax
```
<num-var> = INSTR(<string>,<string>
|,<numeric>|)
```

INT

Integer part

Abbreviation None FUNCTION

Description This function returns the next whole number below the value of the number variable or expression in brackets. In other words the number is truncated.

Examples **INT(1.7) is 1**
INT(–1.7) is –2.
x = INT(Y)

Brackets are optional where sense is not affected.

Syntax <num-var> = **INT** <numeric>

LEFT$

Left string

Abbreviation **LE.** Includes **(** FUNCTION

Description A function which gives the specified number of leftmost characters in a string.

Examples **PRINT LEFT$("ELECTRON",5)**
will give **ELECT**

A$ = LEFT$(B$,C)

Syntax <string-var> = **LEFT$**(<string>,<numeric>)

LEN

Length of string

Abbreviation None FUNCTION

Description A function which gives the number of characters (including spaces) in the specified string.

Examples **PRINT LEN("Donald Duck")**

will be 11

Length = LEN(A$)

Brackets are optional where sense is not affected.

Syntax <num-var> = **LEN**(<string>)

LET

Abbreviation None STATEMENT

Description This is an optional keyword which is used to assign a value to a variable.

Examples **LET X = 1Ø** has same effect as **X = 1Ø**
LET A$ = "JOHN" has same effect as **A$ = "JOHN"**

Comments May not be used with **LOMEM**, **HIMEM**, **PAGE** and **TIME**.

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

LIST

Abbreviation **L** or **FUNC** L COMMAND

Description This command instructs the computer to list the current program on the screen. It has two optional parameters which control the first and last lines to be listed.

Examples **LIST 1ØØ,2ØØ** will list from **1ØØ** to **2ØØ**
LIST ,2ØØ will list up to **2ØØ**
LIST 1ØØ, will list from **1ØØ**

Comments Since **LIST** is a command it cannot be used in a program or in a multi-statement line.

If you press **CTRL** N beforehand, **LIST** will list your program a screen-full at a time. When you want to see the next screen-full, press **SHIFT**. This paged mode can be cancelled using **CTRL** O.

Syntax **LIST**[<num-const>][,][<num-const>]

LISTO

List option

Abbreviation: None — COMMAND

Description: This command must be followed by a number which controls the way in which a program is **LIST**ed, as follows:

0 List just as stored in computer's memory
1 Insert a space after each line number
2 Indent **FOR**... **NEXT** loops
4 Indent **REPEAT**... **UNTIL** loops

Any combination of the above may be obtained by adding the required values.

Examples: **LISTO5** will insert a space after the line number and indent **REPEAT**... **UNTIL** loops.

Comments: Since **LISTO** is a command it cannot be used in a program or in a multi-statement line.

Syntax: **LISTO** <num-const>

LN

Natural logarithm

Abbreviation: None — FUNCTION

Description: A mathematical function to calculate the natural logarithm of the given number variable or expression.

Examples:

```
X=LN(Y)
```

Brackets are optional where sense is not affected.

Syntax: <num-var> = **LN**<numeric>

LOAD

Load program from file

Abbreviation **LO.** or **FUNC** , COMMAND

Description A command which instructs the computer to **LOAD** the named program from the file. If the name is omitted then the next program is loaded.

When the computer prints 'Loading' on the screen, the old program has been deleted and all variables cleared except the resident integer variables.

Examples **LOAD "BUGZAP!"**
LOAD "" loads the next program (from tape only)

Comments During **LOAD**ing, the computer prints up the number of pages of memory being used.

Syntax **LOAD**<string>

LOCAL

Variable declaration

Abbreviation **LOC.** or **FUNC** Q STATEMENT

Description Informs the computer that the named variables are **LOCAL** to the **PROC** or **FN** in which they are declared.

LOCAL variables are totally independent of variables with the same name outside the **PROC** or **FN**.

Examples **LOCAL I**
LOCAL price%

Syntax **LOCAL**<string-var>|<num-var>{,<string-var>|<num-var>}

LOG

Common logarithm

Abbreviation None FUNCTION

Description A mathematical function to calculate the common logarithm of the given number variable or expression.

Examples
```
Y = LOG(X)
rate = LOG(cone)
```

Syntax <num-var> = **LOG**<numeric>

LOMEM

Abbreviation **LOM.** VARIABLE

Description Address pointer containing the address above which all the BASIC program's variables are stored. It is usually set to be the same as **TOP**, but can be altered by the user at the start of a program.

Examples
```
PRINT LOMEM
* LOMEM = &FA2
```

Comments If **LOMEM** is changed during program execution the computer will lose all its BASIC variables.

Syntax
LOMEM = <numeric>
<num-var> = **LOMEM**

MID$

Middle string

Abbreviation **M.** Includes (FUNCTION

Description This function gives a subsection of a string; the position of the first character of the substring and the number of characters being specified. If the length is

omitted, the whole string to the right of the start position is given.

Examples

PRINT MID$(Main$, start, length)

PRINT MID$("MICROCOMPUTER",6,7)
will be **COMPUTE**

PRINT MID$("MICROCOMPUTER",6)
will be **COMPUTER**

X$ = MID$(A$,S,L)

Syntax

<string var> = **MID$**(<string>,<numeric> [,<numeric>])

MOD

Modulo

Abbreviation None

OPERATOR

Description

This function gives the remainder when an integer division is carried out.

Examples

5 MOD 2 is 1
55 MOD 5 is 0
–10 MOD 4 is –2
numerator% MOD denominator%

Comments

If this function is used with decimal numbers or variables, these values are truncated before the division takes place.

4.1 **MOD** 3.9 is 1.

Syntax

<num-var> = <numeric>**MOD**<numeric>

MODE

Graphics mode

Abbreviation **MO.** or **FUNC** M STATEMENT

Description Here is a list of the seven modes and their characteristics:

Mode	Graphics	Colours	Text
Ø	64Ø×256	2	8Ø×32
1	32Ø×256	4	4Ø×32
2	16Ø×256	16	2Ø×32
3	Text only	2	8Ø×25
4	32Ø×256	2	4Ø×32
5	16Ø×256	4	2Ø×32
6	Text only	2	4Ø×25

This instruction tells the computer to change screen mode. Changing mode clears the screen and must not be used within a **PROC** or **FN**. **MODE** resets the value of **HIMEM**.

Examples
```
MODE Ø
MODE x
1Ø MODE mode
```

Comments Text coordinates change from mode to mode, but graphics coordinates are scaled to be the same in all graphics modes: Ø to 1279, Ø to 1Ø23.

Syntax **MODE** <numeric>

MOVE

Move graphics cursor

Abbreviation **MOV.** STATEMENT

Description This instruction moves the graphics cursor to any position, on or off the screen.

Examples
```
MOVE 64Ø, 512
1Ø MOVE X,Y
```

Comments — The graphics origin may be moved to any position on or off the screen by using the instruction

VDU29,X;Y;

where X,Y are the coordinates you wish to become Ø,Ø.

See chapter 2Ø

Syntax — **MOVE** <numeric>,<numeric>

NEW

New program

Abbreviation — None — COMMAND

Description — This command deletes a program from the computer's memory by adjusting certain internal pointers. A program can be retrieved by typing **OLD** which resets the internal pointers, provided no new program lines have been entered and no new variables have been created. Since it is a command it cannot be used in a program or multi-statement line.

Examples

```
NEW
```

Syntax

```
NEW
```

NEXT

Abbreviation — **N.** or **FUNC** N — STATEMENT

Description — This is used to step the control variable in a **FOR . . . NEXT** loop. The control variable after the **NEXT** is optional (see **FOR**).

Examples

```
NEXT X%
NEXT loop
NEXT
```

Syntax — **NEXT** [<num-var>] {, <num-var>}

NOT

Logical NOT

Abbreviation: None — FUNCTION

Description: Normally used in conjunction with a testable condition to reverse the logic of the result, ie **TRUE** becomes **FALSE** and **FALSE** becomes **TRUE**.

Examples:
```
IF NOT(A=5) THEN money = 7Ø.
```

Comments: **TRUE** and **FALSE** are represented as –1 and Ø respectively.

NOTØ is –1, **NOT–1** is Ø.

Beware of trying to use **NOT** with other values for **TRUE**. **NOT1** is –2, which still acts as **TRUE**.

Syntax:
<num-var> = **NOT** <numeric>
<testable condition> = **NOT** (<testable condition>)

OLD

Old program

Abbreviation: **O.** or **FUNC** O (includes **RETURN**) — COMMAND

Description: This command is used to recover a program which has recently been deleted by **NEW**, or by pressing the **BREAK** key, or **CTRL** **BREAK**.

Examples:
```
OLD
```

Syntax:
```
OLD
```

ON

Abbreviation None STATEMENT

Description This instruction can be used in conjunction with **GOTO**, **GOSUB** and **ERROR**.

Firstly **GOTO** and **GOSUB**.

```
ON X GOTO 1ØØ, 3ØØ, 35Ø, 47Ø
```

If X=1 then the program will go to 1ØØ. If X=2 then it will go to 3ØØ. If X=3 then it will go to 35Ø and so on.

```
ON X GOSUB 475, 2Ø5, 31Ø
```

If X=1 then the program will 'gosub' 475. If X=2 it will 'gosub' 2Ø5 and so on.

An **ELSE** can be included at the end to trap out of range values without causing an error.

```
ON X GOTO 7Ø, 19Ø, 31Ø ELSE ENDPROC
```

Secondly **ERROR**

```
ON ERROR GOTO 1ØØØ
ON ERROR RUN
ON ERROR PROCerror
```

This instruction is used for *error trapping*. This enables the program to deal with errors, rather than letting the computer halt the program and print up an error message.

Comments Errors may be accepted once again by typing

```
ON ERROR OFF
```

when the computer will halt and print messages as usual.

Syntax **ON** <num-var> **GOTO** <numeric>{,<numeric>}
or
ON <num-var> **GOSUB** <numeric>{,<numeric>}
or
ON ERROR <statement>
or
ON ERROR OFF

OPENIN

Open input file

Abbreviation **OP.** FUNCTION

Description This function opens a file from the current filing system, eg cassette to the computer and returns the channel number allocated by the computer's file system. If the file does not exist then a **ø** (**FALSE**) is returned. The file is opened for input only. See chapter on file handling.

Examples
```
file = OPENIN "SCREENLOAD"
X = OPENIN A$
```

Syntax <num-var> = **OPENIN** (<string>)

OPENOUT

Open output file

Abbreviation **OPENO.** FUNCTION

Description This function opens a file from the computer to the current filing system, eg cassette and returns the channel number allocated by the computer's file system. If the file does not exist then one will be created. If a file of the same name already exists then it will be deleted and a new one created. The file is opened for output only.

Examples
```
file = OPENOUT "SCREENDUMP"
X = OPENOUT A$
```

Syntax <num-var> = **OPENOUT**(<string>)

OPENUP

Open file for update

Abbreviation None FUNCTION

Description This function opens a file from the current filing system, eg cassette to the computer, in the same way as **OPENIN**. The file is opened for input and output.

Examples
```
edit = OPENUP "Accounts"
Z = OPENUP name$
```

Syntax <num-var> = **OPENUP**(<string>)

OPT

Assembly option

Abbreviation None STATEMENT

Description Used to select whether error messages are reported, or listings are given during assembly of a machine-code subroutine. **OPT** can only be used inside the square brackets enclosing a piece of Assembly Language.

It may take eight different values of parameter:

OPT 0 Report no errors, list no machine-code
OPT 1 Report no errors, list the machine-code
OPT 2 Report any errors, list no machine-code
OPT 3 Report any errors, list the machine-code

OPT 4 to **OPT 7** are the same as **OPT**s 0 to 3 except that the machine-code is generated at the origin O% instead of the program counter P%.

Examples
```
50 OPT 2
100 For I = 0 TO 3 STEP 3
110 [OPT I
120 ....
130 ....]
140 NEXT
```

Comments — The second example above is commonly used in two-pass assembly, which will always give errors on the first pass.

Syntax — **OPT** <numeric>

OR

Logical OR

Abbreviation — None — OPERATOR

Description — This is a logical operator which combines two testable conditions in the following way:

False **OR** False gives False
False **OR** True gives True
True **OR** False gives True
True **OR** True gives True

Examples

```
IF A = 5 OR A = 10 THEN . . .
IF Altitude OR B% THEN . . .
IF X < 2 OR NOT(Y=4) THEN . . .
```

Syntax — <num-var> = <numeric> **OR** <numeric>

OSCLI

Operating system command line interpreter

Abbreviation — **OS.** — STATEMENT

Description — Used to pass BASIC variables as parameters to operating system calls, such as ***FX** or ***KEY**. Must be used with a string, the variables being converted using **STR$**. The complete string must be the same as would be used normally, but without the asterisk.

Examples

```
OSCLI"FX"+STR$ action+","+STR$ parameter
OSCLI"KEY"+STR$ keynumber+keystring$
```

PAGE

Abbreviation **PA.** VARIABLE

Description Address pointer containing the address above which the BASIC program is stored. It is usually set to &EØØ on cassette only machines, but can be altered by the user to locate more than one BASIC program in memory.

Examples
```
PRINT PAGE
PAGE = &16ØØ
```

Comments The two least significant hex digits of **PAGE** are always zero – it points to the base of a page of memory, which always contains 256 bytes.

Syntax
PAGE = <numeric>
<num-var> = **PAGE**

PI

Abbreviation None CONSTANT

Description **PI** has the value 3.14159265 and can be used just like a number.

Examples
```
circumference = 2*PI*radius
area = PI*radius ∧2
```

Syntax <num-var> = **PI**

PLOT

Plot graphics

Abbreviation **PL** or **FUNC** P STATEMENT

Description The **PLOT** instruction is used to draw single points, lines, dotted lines and triangles.

PLOT takes the form:
PLOT A,X,Y

which will plot at or to the point X,Y in the manner determined by the value of the first parameter A.

The effect of the value of the first parameter is:

Ø 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 A have other effects which are related to the effects given by the values Ø to 7

8-15 As Ø-7 but with the last point in the line omitted in 'inverting actions' – eg using GCOL4.

16-23 As Ø-7 but with a dotted line.

24-31 As Ø-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 Ø-7 but only a single point is plotted.

72-79 Line fill.

8Ø-87 As Ø-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 Line fill.

See chapter 2Ø

Syntax **PLOT** <numeric>, <numeric>, <numeric>

POINT

Point at graphics screen colour position

Abbreviation **PO.** FUNCTION

Description This function gives the logical colour at the specified point on the screen. If this point is off the screen then a –1 is given.

Examples **colour = POINT(X,Y)**

Comments This function is used in the Marslander program to test whether the capsule has touched down, and whether it is on a flat surface.

Syntax <num-var> = **POINT** (<numeric>, <numeric>)

POS

Position of text cursor

Abbreviation None FUNCTION

Description This function tells how far across the text screen the text cursor is . Ø is on the left, and the value to the right is determined by the mode, and by the size of text window.

Examples **X = POS**

Syntax <num-var> = **POS**

PRINT

Print on screen

Abbreviation **P.** or **FUNC /** STATEMENT

Description This instruction is used for all character output to the screen.

Examples
```
PRINT "Anywhere"
PRINT A,B,length,moon$,BILL%
PRINT HEIGHT*DEPTH;CHR$127; 99;
```

PRINT CHR$X; is almost the same in operation and effect as **VDUX**, and the two are interchangeable.

Comments **PRINT** is also used to issue control-codes to the computer during program execution.

PRINT CHR$9 will move the text cursor forward one square, for example.

Syntax **PRINT** {["][,|;]<string>|<numeric>}["][;]

PRINT#

Print on file

Abbreviation **P.#** STATEMENT

Description **PRINT#** is like **PRINT** but instead of printing information to the screen it prints it on to a previously opened file. Must be followed by channel number (see chapter on file handling).

Examples
```
PRINT# channel, make$, prices
PRINT# C,B,A,Z$
```

Syntax **PRINT#**<num-var>{,<numeric>|<string>}

PROC

Procedure

Abbreviation **FUNC** X — Prefix

Description This prefix is used when defining a named procedure (see **DEF**) and to call this named procedure from anywhere in the program.

Examples
```
PROCrotate
IF . . . THEN PROCfire
DEF PROCfire
PROCvolume(radius,height)
```

Comments **PROC** must be followed by a name without any spaces. Parameters may be passed in brackets.

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

PTR#

Reserved for future use.

RAD

Radians

Abbreviation **FUNC** J — FUNCTION

Description This function gives an angle in radians which is equivalent to the specified angle in degrees. There are **2*PI** radians in a circle of 36Ø°.

Examples
```
X = RAD(X)
angle = RAD(angle)
answer = SIN(RAD(angle))
```

Brackets are optional where sense is not affected.

Syntax <num-var> = **RAD** <numeric>

READ

Read data into variable(s)

Abbreviation None STATEMENT

Description This instruction tells the computer to copy information from a **DATA** statement into the variables which follow the **READ** instruction. The types of variables must match; numbers must be copied into numeric variables and strings into string variables. See **DATA** and chaper on **READ** and **DATA**.

Examples
```
READ name$, Tel, credit
```

Syntax **READ** <num-var>|<string-var>{,<num-var>|<string-var>}

REM

Remark

Abbreviation None STATEMENT

Description This instruction tells the computer to ignore the rest of the program line, thus enabling the programmer to insert comments in the listing without affecting the program.

Examples
```
10 REM Marslander
1035 REM Move Alien
```

Syntax **REM** <anything>

RENUMBER

Renumber program line

Abbreviation **REN.** or **FUNC** 8 COMMAND

Description This command has two optional parameters which control the way in which a program is to be renumbered. The value of the first parameter is the starting line number. The second is the increment for

each subsequent line number. If either or both parameters are omitted they will default to 1Ø.

Examples

If a program looks like this

```
1Ø xxx
15 xxx
17 xxx
3Ø xxx
32 xxx
```

RENUMBER 1ØØ,5 will change it to

```
1ØØ xxx
1Ø5 xxx
11Ø xxx
115 xxx
12Ø xxx
```

Comments

This command also renumbers all **GOTO**s, **GOSUB**s and any other cross references, except calculated ones like **GOTO (8*W)** and **GOSUB (1ØØ+Z)**.

Syntax

RENUMBER [<num-const>[,<num-const>]]

REPEAT

Abbreviation **REP.** or **FUNC** Y STATEMENT

Description

This instruction initiates a **REPEAT . . . UNTIL** loop which tells the computer to **REPEAT** a set of instructions **UNTIL** a testable condition becomes true.

REPEAT . . . UNTIL loops may be nested up to a depth of 2Ø.

Examples

```
NOW = TIME: REPEAT UNTIL TIME -NOW >= 5ØØ
```

Syntax

REPEAT

REPORT

Abbreviation **REPO.** STATEMENT

Description This instruction will print up on the screen what the last error was, in words.

Examples
```
REPORT: PRINT " at line ";ERL
```

Syntax **REPORT**

RESTORE

Abbreviation **RES.** or **FUNC** W STATEMENT

Description This instruction restores the '**DATA** pointer' to the beginning of a specified line. The **DATA** pointer points to the next piece of information to be read by a **READ** instruction. If no line is specified the **DATA** pointer is restored to the beginning of the first **DATA** instruction in the program.

Examples
```
RESTORE
RESTORE 2500
```

Syntax **RESTORE** <numeric>

RETURN

Return from subroutine

Abbreviation **R.** STATEMENT

Description This instruction informs the computer that it has reached the end of a subroutine and that it must now **RETURN** to the instruction immediately after the **GOSUB** at which the subroutine was called. A subroutine must not be exited other than by using **RETURN**.

Examples
```
RETURN
IF A=0 THEN RETURN
```

Comments Not to be confused with the **RETURN** key.

Syntax **RETURN**

RIGHT$

Right string

Abbreviation **RI.** Includes (FUNCTION

Description A function which gives the specified number of right-most characters in a string.

Examples **PRINT RIGHT$("FLAVOUR", 3)** will give OUR.

Syntax <string-var> = **RIGHT$**(<string>,<numeric>)

RND

Random number generator

Abbreviation None FUNCTION

Description This function, which may be followed by a number in brackets, returns a random number.

RND by itself generates a random whole number between –2147483648 and 2147483647

RND(–X) gives 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 **X = RND(1Ø)**

Syntax <num-var> = **RND**[(<numeric>)]

RUN

Abbreviation **FUNC** R (includes **RETURN**) STATEMENT

Description This instruction makes the computer initiate execution of the numbered program lines in its memory. It also clears all variables except the resident integer variables.

Examples
```
RUN
IF velocity >1ØØ THEN RUN
```

Comments A program can be **RUN** without clearing the variables by using the command **GOTO**, followed by the first line number.

Syntax **RUN**

SAVE

Abbreviation **SA.** or **FUNC** . STATEMENT

Description This transfers the program from the computer's current program area (between the system variables **PAGE** and **TOP**) on to cassette, and in future expansions, disc as well. When used with tape, **SAVE** must be followed by a name of up to 1Ø characters, inside quotation marks.

Examples **SAVE "BUGZAP!"**

Syntax **SAVE** <string>

SGN

Sign

Abbreviation None FUNCTION

Description This function tells you whether the specified number, variable or expression is positive, zero or negative.

The function gives:

–1 for a negative number
Ø for a zero
+1 for a positive number

Examples **X = SGN(Y)**

Brackets are optional where sense is not affected.

Syntax <num-var> = **SGN**(<numeric>)

SIN

Sine

Abbreviation None FUNCTION

Description This function gives the sine of an angle, which must be in radians.

Examples **PRINT SIN(3.142)**
X = SIN(y)

Brackets are optional where sense is not affected.

Syntax <num-var> = **SIN**(<numeric>)

SOUND

Abbreviation **SO.** STATEMENT

Description Makes the computer generate a sound on the internal loudspeaker.

The format is:

SOUND Q,A,P,D

Q is the channel number, Ø to 3.

A is the envelope number, 1 to 4. If A is Ø then that sound channel is turned off. If A is negative (–1 for example) then a pure tone is produced.

P is the pitch, Ø to 255.

D is the duration, 1 to 255 in twentieths of a second.

See chapter 22

Syntax **SOUND** <numeric>,<numeric>,<numeric>, <numeric>

SPC

Space

Abbreviation None (See PRINT, INPUT)

Description This statement is used in conjunction with **PRINT** or **INPUT** to give the specified number of spaces. This number may not be greater than 255.

Examples
```
PRINT "Name";SPC(1Ø);"Tel.";SPC(1Ø);"CREDIT"
INPUT "Amount" SPC(3) A
```

Brackets are optional where sense is not affected.

Syntax
PRINT SPC (<numeric>)
INPUT SPC (<numeric>)

SQR

Square root

Abbreviation None FUNCTION

Description This function gives the square root of a positive number.

Examples
```
X = SQR(Y)
ans = SQR(ABS(NUMBER))
```

Brackets are optional where sense is not affected.

Syntax <num-var> = **SQR**(<numeric>)

STEP

Abbreviation **S.** or **FUNC** S (See FOR)

Description This is part of the **FOR . . . TO . . . STEP . . . NEXT** statement. The number following **STEP** is the amount by which the control variable is incremented in each loop, and can be positive or negative, integer or real.

Examples
```
FOR X = 0 TO 50 STEP 5
FOR I = 10 TO 0 STEP -1
FOR J = 7 TO 11.3 STEP Z*.7
```

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

STOP

Abbreviation None STATEMENT

Description This instruction has the same effect as **END** but it prints the message **STOP**, and the line number, on the screen.

Example
```
IF P >Temp THEN STOP
```

Syntax **STOP**

STR$

String

Abbreviation **STR.** FUNCTION

Description This function converts any number or expression in the brackets into a string. **STR$** has an opposite effect to that of **VAL**.

Examples
```
A$ = STR$(X)
B$ = STR$(-1.23)
```

Brackets are optional where sense is not affected.

Syntax <string-var> = **STR$** (<numeric>)

STRING$

Abbreviation **STRI.** Includes (FUNCTION

Description This instruction produces a long string consisting of a specified number of copies of a shorter string.

Examples
```
Line$ = STRING$(40,"-")
```

Syntax <string-var> = **STRING$** (<numeric>,<string>)

TAB

Abbreviation None (See PRINT, INPUT)

Description Used with either **PRINT** or **INPUT** to set the position of the text cursor on the screen.

TAB(X) will move the cursor forward to position X on the current line. X can be between Ø and 19, Ø and 39, or Ø and 79 depending upon which mode is in use.

TAB(X,Y) will move the cursor directly to position X,Y. The text cursor has its origin Ø,Ø at the top left. If a text window is in use, then **TAB** will treat the top left corner of that as Ø,Ø.

Examples

```
PRINT TAB(6,11); "HELLO"
INPUT "How much" TAB(1Ø), cost
```

Comments — The brackets are compulsory, and there must be no space between **TAB** and (.

Syntax

PRINT TAB(<numeric> [,<numeric>])
INPUT TAB(<numeric> [,<numeric>])

TAN

Abbreviation — **T.** — FUNCTION

Description — This function gives the tangent of an angle, which must be in radians.

Examples

```
PRINT TAN(PI/2)
ratio = TAN(.6)
```

Brackets are optional where sense is not affected.

Syntax — <num-var> = **TAN**<numeric>

THEN

Abbreviation — **TH.** or **FUNC** T — (See IF)

Description — Provides the course of action if the result of an **IF** statement is true.

It is an optional keyword, but the program is neater and easier to follow when it is left in.

Examples **IF A = B THEN PROCab**

Comments **THEN** is not optional when assigning certain resident variables such as **TIME**, and when **GOTO** is omitted.

Syntax **IF**<testable condition> **THEN**<statement>[**ELSE** <statement>]

TIME

Abbreviation **TI.** VARIABLE

Description An integer variable which is incremented every hundredth of a second. It serves as an elapsed time clock, and can be set to any initial value by the user.

Examples
```
TIME = 0
T% = TIME
PRINT T%
```

Syntax
```
TIME = <numeric>
<num-var> = TIME
```

TO

Abbreviation None (See FOR)

Description Used in the **FOR . . . NEXT** loop to set the limiting value of the control variable.

Examples **FOR I = 0 TO 11**

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

TOP

Abbreviation None VARIABLE

Description Address pointer containing the address of the first free memory location after the top of the BASIC program. **TOP–PAGE** will give the length of your BASIC program in bytes.

Examples
```
PRINT TOP–PAGE
```

Syntax <num-var> = **TOP**

TRACE

Abbreviation **TR.** STATEMENT

Description Debugging device which prints BASIC line numbers in order of execution. Is turned on by **TRACE ON**, and off by **TRACE OFF**. **TRACE X** will only give line numbers below X.

Syntax **TRACE ON** | **OFF**<numeric>

TRUE

Abbreviation None CONSTANT

Description This is a condition which the computer understands to be the number –1. If the computer decides a certain condition is true, it will represent it as –1, and will act accordingly.

Examples
```
IF A = TRUE THEN . . .
Test = TRUE
```

Comments — In practice, any number other than **Ø** is taken by the computer to be **TRUE**. Care is needed though. **NOT(–1)** is **Ø**. **NOT(1)** is –2. This is because the **NOT** function simply inverts the binary digits, and does not consider them true or false at all.

Syntax — <num-var> = **TRUE**

UNTIL

Abbreviation — **U.** or **FUNC** U (See REPEAT)

Description — Conditional part of **REPEAT . . . UNTIL** loop. The loop is executed until the conditional statement after **UNTIL** goes true.

Examples

```
REPEAT
xxx
xxx
UNTIL X = 7
```

```
REPEAT          These are       REPEAT
xxx             the same        xxx
xxx             and will        xxx
UNTIL Ø         loop forever    UNTIL FALSE
```

Syntax — **UNTIL** <testable condition>

USR

Abbreviation — None FUNCTION

Description — Used from BASIC to call a previously assembled machine-code function. Similar in operation to an **FN**, but is not able to pass parameters. Used in preference to an **FN** where long calculation is involved, and speed is at a premium.

Examples **X = USR(&175Ø)**

```
2Ø address = &3ØA9
3Ø PRINT USR(address)
```

Comments See chapter on assembler.

Syntax <num-var> = **USR**(<numeric>)

VAL

Abbreviation None FUNCTION

Description Gives the numeric part of a string as a number. The string must start with +, –, or a number, otherwise Ø is given.

Examples **number = VAL(–762*12)**

will put number equal to **–762**.

Syntax <num-var> = **VAL**(<string>)

VDU

Abbreviation **V.** or **FUNC V** STATEMENT

Description **VDU** has almost the same function as **PRINT CHR$**. It can be used to give any character or control code from the ASCII table in Appendix F.

Examples **VDU5** Link text and graphics cursors.
VDU8 Move text cursor back one square.
VDU23 Re-define character.

Comments Sends code directly to the **VDU** drivers. Is quicker to type than **PRINT CHR$**.

PRINT TAB(X,Y) is equivalent to
VDU31,X,Y

Syntax **VDU** <numeric> {, | ; <numeric>} [;]

VPOS

Abbreviation **VP.** FUNCTION

Description Gives the distance of the text cursor from the top of the screen or text window.

Examples **Y = VPOS**

PRINT VPOS

Syntax <num-var> = **VPOS**

WIDTH

Abbreviation **W.** STATEMENT

Description Sets the width used by the computer to print on the screen. This is normally unlimited, and the computer runs on to the next line at the right boundary of the screen or text window.

Examples **WIDTH 10** will cause the text to be printed in a column 10 characters wide.

Syntax **WIDTH** <numeric>

26 Cassette file handling

Aside from saving programs, data files can be opened on the cassette: for example to store addresses or telephone numbers.

Here is a list of file handling commands which you can use:

***CAT**	Gives a catalogue of all data files and programs on the cassette. It takes a very long time on a cassette.
OPENIN	Opens a file so that it can be read.
OPENOUT	Opens a new (empty) file for writing.
OPENUP	Opens a file for reading or 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.
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.

To create a data file, you must first open it using **OPENOUT**. **OPENOUT** must be assigned to a variable, as follows:

A = OPENOUT "stamps"

In this case, the file called 'stamps' has been opened, and is allocated to a variable called A. A becomes the communication channel to the file, and all data sent to the file is routed via A. For example, if you want to write the names of all your stamps into the file, you use **PRINT#**:

```
10 A = OPENOUT "stamps"
20 REPEAT
30 INPUT "Give the name of the stamp",name$
40 PRINT# A, name$
50 UNTIL name$ = "NO MORE"
60 CLOSE# A
```

So after the file has been opened, its name is not mentioned again. The above program will allow you to make a list of names and, if the cassette player is switched on, they will be recorded on tape. Notice that the file must be closed after use, with **CLOSE#**.

To get the data back into the computer, you must open the file for reading using **OPENIN**. (The tape must be wound to the start of the file, and the PLAY button pressed).

```
A = OPENIN "stamps"
```

The variable name, in this case A, is completely arbitrary. You could equally well call it FRED, file, or anything else you wish. To read data from the tape into the computer's memory, use **INPUT#**:

```
10 A = OPENIN "stamps"
20 REPEAT
30 INPUT# A, name$
40 UNTIL name$ = "NO MORE"
60 CLOSE# A
```

Line 40 could equally well read:

```
40 UNTIL EOF# A
```

EOF# is a logical file operator which is **TRUE** when the end of the file has been reached.

PRINT# and **INPUT#** are used to write or read strings to and from the cassette file. The instructions **BPUT#** and **BGET#** are used to write and read single characters.

***CAT** may be used anytime to give a catalogue of all program and data files on tape, and *. can be used as an abbreviation.

Cassette operations print messages on the screen, and sometimes cassette operations will produce errors. The message printed, and the computer's reactions to errors can be altered using ***OPT**:

***OPT1,**X controls all but the error messages which are printed on the screen.

X = Ø Gives no messages.
X = 1 Gives short messages (as normal).
X = 2 Gives long messages, including load and execution addresses.

***OPT2,**X controls the computer's action.

X = Ø Lets the computer ignore all errors, and carry on regardless. Messages can still be given.
X = 1 The computer asks you to try again by rewinding the tape (as normal).
X = 2 The computer aborts the operation.

***OPT3,**X sets the inter-block gap in tenths of a second. This only applies to **PRINT#** and **BPUT#**. The gap on **SAVE** is fixed to Ø.6 seconds.

***OPT** on its own sets all the values to normal.

27 Error handling

So far you have seen that when the computer finds an error it halts execution of the program and prints a message on the screen. Some errors are generated by incorrect programming and these are the ones that you have to correct. But what about errors which occur during the execution of a good program, because either the data is wrong or the user inputs something that the computer cannot handle?

Look at the following program:

```
10 REPEAT
20 INPUT "NUMBER", N
30 L = LOG(N)
40 PRINT "LOG OF ";N;" IS ";L
50 UNTIL FALSE
```

This is simple enough. It takes a number from the keyboard, and gives you the log of that number. But, if you type in a negative number, the program comes to a halt with:

```
Log range at line 30
```

The same thing happens if you type 0, or a character such as W, or a word such as TWELVE.

It is easy to trap such an error, and to print a message to tell the user what he or she has done wrong. Every error has an error number, which is stored in a variable called ERR. You will find a list of these errors in Appendix B. The number for the log error is 22, so we can alter the program as follows:

```
5 ON ERROR GOSUB 100
10 REPEAT
20 INPUT "NUMBER", N
30 L = LOG(N)
40 PRINT "LOG OF ";N;" IS ";L
50 UNTIL FALSE
100 IF ERR = 22 THEN PRINT "MUST BE A POSIT
IVE NUMBER > 0"
110 RETURN
```

Now, if an illegal input is made, the program simply prints a message telling you what you have done wrong, and carries on running. The trouble is, it carries on running, and running, and running. The **ESCAPE** key has no effect. This is because **ESCAPE** is treated as an error; it even has its own error code: 17. You'll have to press **BREAK**, then **OLD** **RETURN** to get the program back.

To overcome this, **RETURN** can be incorporated into line 1ØØ:

```
100 IF ERR = 22 THEN PRINT "MUST BE A POSIT
IVE NUMBER > 0":RETURN
```

Now, the **ESCAPE** key works, but no message is printed to say what happened, or at what line. Don't forget to delete line 11Ø!

To print the message, the instruction **REPORT** is used.

```
110 REPORT
```

will print the message **Escape** on the screen.

The line number at which errors occur is stored in a variable called **ERL**. If you add a line 12Ø:

```
120 PRINT " at line ";ERL
```

then the correct message will be printed up on the screen.

The above example is fairly simple, because there is only the one error which can occur. If you write a program that other people will use, you will have to think of all the possible errors that may occur, and trap them accordingly. In the example, the instruction used was

```
ON ERROR GOSUB
```

Other useful error trapping instructions are:

```
ON ERROR GOTO
ON ERROR PRINT
ON ERROR PROC
```

If you do use **GOTO**, you cannot 'GOTO' back again into a **FOR . . . NEXT** loop, a **REPEAT . . . UNTIL** loop, or a function. See Appendix B for a list of errors and their codes.

28 Merging BASIC programs

Two methods are given here by which you can merge two BASIC programs which are stored on cassette, and in future expansions, disc as well.

(i) This method requires you to **LOAD** one of the programs (preferably the shorter) and then to re-save it as an ASCII file using ***SPOOL**. You can then **LOAD** the other program; and the spooled program will be entered as the equivalent of keyboard input by loading it using ***EXEC**. Assuming that the two programs are called **LARGE** and **SMALL**, the procedure is as follows:

```
LOAD "SMALL"
```

Now set the tape recorder to a blank section of tape.

```
*SPOOL "SMALL"
LIST
*SPOOL
```

The program **SMALL** is now saved as an ASCII text file.

```
LOAD "LARGE"
*EXEC "SMALL"
```

The merger is now complete. Any line numbers in **LARGE** that coincide with those in **SMALL** will be overwritten. If you want to add **SMALL** to the end of **LARGE** then you have to adjust the line numbers before **SMALL** is spooled. When you use ***SPOOL**, anything that is output to the screen is also sent to the cassette. That is why you must type **LIST**. ***SPOOL** without the file name closes the file that has been spooled.

(ii) This is a slightly simpler method, but the line numbers of **SMALL** must be adjusted so as all to be higher than the highest line number in **LARGE**. The idea is to **LOAD** the program with the lower line numbers, and ***LOAD** the program with the higher numbers at **TOP–2**. Lastly, it is necessary to type **END** so that the computer can trace the lengthened program.

The procedure is:

```
LOAD "LARGE"
OSCLI "LOAD ""SMALL""" + STR$~(TOP-2)
END
```

This method is very easy, but you must be careful to adjust the line numbers.

29 Assembly Language

Introduction

The computer's 'brain' has its own language, and that language is not BASIC. Every time you run a BASIC program, each line has to be translated before this brain (the computer's *central processor unit*) can understand it at all. This translation is accomplished by a device called an *interpreter*, which resides in the computer's memory. The action of this device need not concern you, but it is itself a program written in *machine-code*, and machine-code is the computer's own language.

There are 55 different instructions in machine code, which is about half the number of BASIC instructions available on the Electron. Each of these instructions acts upon one or more of the registers inside the 65Ø2 microprocessor (65Ø2 is the type-number of this processor – it has no significance). A register is just like a byte of memory. The 65Ø2 contains six registers, five of them being 1 byte long, and the last being 2 bytes long. These registers are not a part of the computer's memory map (from location &ØØØØ to &FFFF); they live an entirely separate existence in the heart of the microprocessor. But the machine-code instructions which control these registers are stored in the computer's memory, in the position on the memory map labelled 'operating system'. These instructions don't look much like intelligible commands, for they are simply binary numbers – 1Ø1Ø1ØØ1ØØØØ1Ø1Ø for example. It is very difficult to program using such low-level instructions; even in hexadecimal they hardly look any better: A9 ØA. This is the reason for using *Assembly Language*.

Assembly Language uses a three-letter mnemonic to represent each machine-code instruction. Each mnemonic is a contraction of the action-in-words of that instruction.

Take the instruction given above. One of the registers in the microprocessor is called the accumulator (all the registers will be described in detail in a moment).

A9 means 'load the accumulator'.

The mnemonic for this is LDA, thereby giving you a rough guide to its function, LoaD Accumulator.

The other part of the instruction, ØA, is 1Ø in decimal. So A9 ØA means 'put 1Ø in the accumulator', and this is written in Assembly Language as:

```
LDA #1Ø
```

(The hash (#) tells the computer that it is the 1Ø which is to be put into the accumulator, and not the contents of memory location 1Ø. This will be explained in a moment.)

So, each of the 55 machine-code instructions is assigned a three-letter assembly mnemonic, which enables you the programmer to understand the function of each, without having to look it up on a chart.

The Electron has another program in its memory, called an *assembler*, and this converts the Assembly Language directly into machine-code. During this assembly process, the computer can help you by giving error messages and a listing of the machine-code in hex. (If you were programming the 65Ø2 direct in machine-code there would be no error messages at all – and just try finding a mistake among a few hundred machine-code instructions!)

The assembler loads the machine-code into memory, and it can then be run, either as a **CALL** or **USR** from BASIC, or by using ***RUN**.

Registers in the 65Ø2

The 65Ø2 microprocessor has six registers as follows:

Accumulator
The accumulator is the main working register of the processor. Most of the 55 Assembly Language instructions operate on the accumulator, which gained its name from the way that results of arithmetic operations are 'accumulated'. It is an 8-bit register, meaning that it can store and operate upon eight binary digits (one byte). Each bit is designated a number, from Ø for the least significant (rightmost) to 7 for the most significant (leftmost).

Common operations involving the accumulator are:

- Loading it from memory (the locations &ØØØØ to &FFFF).
- Storing its contents in memory.
- Addition or subtraction.
- Logical functions (AND, OR, or EOR).
- Shifting its contents left or right.

Index registers X and Y

The two index registers are each 8-bits long, and are used for the following:

- To be added to the address used by an instruction. This is called *indexing*.
- As general purpose registers for various counting or short term memory duties.
- In addition to the above, both the accumulator and the two index registers are used by the Electron to pass parameters to operating system subroutine calls. This will be explained later.

Program counter

The program counter is the only 16-bit register, and it holds the memory address of the next instruction to be executed.

Operations involving the program counter are:

- Jump and branch instructions which alter the contents of the PC and thereby divert the flow of the program. (Much like **GOTO** in BASIC.)

Stack pointer

The stack pointer is an 8-bit register, with a ninth bit on the most significant end which is always set to 1. It is an address pointer which gives the location in memory of a special kind of data-structure used by computers called the *stack*. It can point to addresses between &Ø1ØØ and &Ø1FF. The stack is explained later, but in essence it is a section of memory which has not only a position, but also an order. Thus, data which is pushed on to the stack in one order, can only be pulled off it in the reverse order. This sort of memory is called *last in first out* (LIFO). It is used for storing data in which the order is important, eg execution addresses of nested subroutines.

Flags register

The flags register is different from all the others in that it operates as seven single-bit registers: N, V, B, D, I, Z, C. Each bit signals a condition in the processor, and certain instructions act upon these conditions (whether that condition is present, true; or is not present, false).

Each bit acts as follows:

Bit N is set to 1 when the last operation produced a negative result. A negative result is signified by the most significant bit of a register being 1 (the sign bit). In the case of the accumulator, a result inside it of, for example, 1ØØ1Ø1ØØ would set bit N of the flags register to 1. If the last operation did not produce a negative result then bit N is reset to Ø.

Bit V is set to 1 when the last operation overflowed into the sign bit. As stated above, the sign bit is bit 7 in the case of the accumulator, so bit V is set to 1 when there is a carry from bit 6 to bit 7. This is important to know when using twos complement arithmetic, for it means that an error has occurred which must be corrected.

Bit B is set to 1 when the **BRK** command is used (break). (This command has much the same effect on a machine-code program that **ESCAPE** has on a BASIC program.)

Bit D, when set to 1, causes the processor to operate in BCD mode (binary coded decimal). When reset to Ø, the processor works as normal in binary. BCD is beyond the scope of this book, and need not concern you.

Bit I is the interrupt mask. When it is set to 1 no interrupts are accepted. Interrupts are also beyond the scope of this book.

Bit Z is set to 1 when the last operation produced a zero result.

Bit C is the carry register. It is set to 1 by a carry from the most significant bit of one of the registers, usually the accumulator.

These flags are used by the branch instructions, which direct the flow of the program according to the conditions. For example, BEQ means 'branch if equal to zero'. The program will branch if the Z bit is set to 1. If not it will not branch.

Addressing modes

Take a single instruction – you have seen **LDA** before. Its function is always to 'load the accumulator', but it may load it in different ways and from different places according to which *addressing mode* is used.

```
LDA #1Ø
```

means 'load the accumulator with 1Ø'. You know that already. However,

```
LDA 1Ø
```

means 'load the accumulator with the contents of memory location 1Ø'.

This is an example of two different addressing modes. The first is *immediate addressing*. The instruction uses the data immediately, without looking for it in memory. The second is *zero-page addressing*. The instruction uses the contents of the address specified. It is called zero-page because the computer's memory is divided up into 256 pages each of 256 bytes. Any address which has its two most significant hex digits as zero is said to be in the zero-page of memory. The zero-page extends from locations &ØØØØ to &ØØFF.

LDA may also be used with a full 16-bit address:

```
LDA &3ØA7
```

will 'load the accumulator with the contents of memory location &3ØA7'. This addressing mode is called *absolute*. It can access any location in the computer's memory. Notice that the assembler treats numbers as decimal, unless they are preceded by &.

Immediate, zero-page and absolute are not the only addressing modes, although they are the most simple to understand.

```
LDA &1D77,X
```

is an *indexed addressing* mode.

The address used by the instruction is &1D77 plus the contents of index register X. So the accumulator is not loaded from &1D77 but from &1D77+X. Note that the contents of index register X are added to the address, and not to its contents.

The index register used can equally well be Y:

```
LDA &2500,Y
```

(Note: When using machine-code there are several subdivisions of the above indexed addressing mode, but using the assembler takes care of all those for you. However, the assembled machine-code (in hex) will not always be the same for the same indexed instruction.)

Another still more complicated addressing mode is *indirect addressing:*

```
LDA (&1B,X)
```

The address given after the assembler mnemonic, in this case &1B, must be a location in the zero-page of memory (or an error will result). This location is then added to the contents of the X register, to give another location in the zero-page. The contents of this new location, and the contents of the location above it, together supply the full 16-bit address of the location from which the accumulator is loaded. So, if &1B+X contains &AA, and &1B+X+1 contains &BB, then the accumulator will be loaded with the contents of memory location &AABB.

The above operation is called *pre-indexed indirect addressing;* the indexing is the addition of the X register, and the direction is the use of the two consecutive locations at the intermediate address as an address pointer to the actual location used. It is called pre-indexed indirect because the indexing is done before the indirection. All pre-indexed indirections must use index register X.

Post-indexed indirect addressing is written in Assembly Language as follows:

```
LDA (&27),Y
```

In this addressing mode, the indirection occurs first. The address given after the assembler mnemonic, in this case &27, must again be a location in the zero-page of memory. The contents of this location and the contents of the location above it together give a 16-bit address. To this 16-bit address is added the contents of index register Y, and this final address is the location from which the accumulator is loaded. All post-indexed indirections must use index register Y.

The above examples show the complete range of addressing modes which can be used with the instruction **LDA**. However, there are three more important addressing modes which are used with certain other instructions.

All of the branch instructions use a *relative addressing* mode. **BEQ** was mentioned in the description of the flags register; it means 'branch if equal to zero'. A branch is an instruction which has an *offset:*

```
        ZZZ data
        BEQ Label
        AAA data
        BBB data
.Label  CCC data
        DDD data
```

In this fragment of program, the triple-letters can be any assembler instructions. When a program is running, the program counter is incremented one step at a time to point at the next location which is to be executed. In this example, when **BEQ** Label is being executed the program counter will point to the line containing the instruction marked **AAA**. If the result of checking the Z flag is that the previous operation did not produce a negative result then execution will continue at the line containing **AAA**. If the previous operation did give a zero result then the program counter is incremented until it points at the line marked Label. This program illustrates the use of labels in assembler. They can take any name that you choose (subject to the same limitations as a BASIC variable name), and are signified by the fact that they must always start with a full stop.

Branch instructions may branch to labels either forwards or backwards, but not too far. The actual distances are 128 bytes backwards or 127 bytes forwards; but remember that these are measured from the next instruction following the branch, and that each instruction may be either 1, 2, or 3 bytes long. The assembler will soon tell you if you have an address or label out of range.

The next addressing mode is *accumulator addressing*, which is used by only four instructions in the 65Ø2 set. These are **ASL**, **LSR**, **ROL**, and **ROR**, and their action is explained in the reference section. In essence, they shift the bits of a memory location or the accumulator to the left or right.

ASL &76Ø

means shift the contents of memory location &76Ø one bit to the left. In order to apply this instruction to the accumulator, the accumulator's own addressing mode is used:

ASL A

means shift the contents of the accumulator one bit to the left. Look up the four instructions in the reference section for more information.

The final addressing mode which you need to know about is the simplest. Certain instructions, such as **BRK** (break) do not need any data or memory reference at all. These are called *implied* instructions and they carry out a simple task, usually on one of the registers; for example **CLC** meaning 'clear the carry flag'.

Addressing mode	Examples	
Immediate	**LDA #68**	**LDA #number**
Zero-page	**LDA &9B**	**LDA address**
Absolute	**LDA &8E17**	**LDA address**
Indexed	**LDA &AØ6C,Y**	**LDA table,X**
Pre-indexed indirect	**LDA (&72,X)**	**LDA (pointer,X)**
Post-indexed indirect	**LDA (&ØØ),Y**	**LDA (zero),Y**
Relative	**BEQ Repeat**	**BNE Loop**
Implied	**CLC**	**BRK**
Accumulator	**LSR A**	**ROL A**

The examples on the right show the assembler mnemonics used not with specific addresses, but with BASIC variables. You will find out that this is a good way of writing assembler subroutines which are to be called from within BASIC programs by **CALL** or **USR**.

One final point about addressing modes. The **JMP** (jump) instruction is the only one which allows straight indirect addressing (non-indexed). **JMP** is very similar to BASIC's **GOTO**. It can take a full 16-bit address and place this value in the program counter – hence the program jumps to a new execution address. It is usually used with a label, just like branch, but without the restriction on distance. In absolute mode it would look like this:

JMP Label

If you wish to use it in indirect mode then simply enclose the address in brackets:

JMP (&21A7)

It will then use the contents of the two consecutive locations at &21A7 as an address pointer to the location to which it will jump.

```
          JMP &21A7
            .
            .
            .
&21A7     &32
          &76
            .
            .
            .
&3276   · continue execution.
```

Entering assembly mnemonics

This section tells you how to write Assembly Language subroutines, and how to call them from BASIC. You may find it worthwhile, now that you know about the 65Ø2 processor's make-up, to read all of the assembly mnemonic definitions. You will then be able to understand much more clearly the capability of the processor, and what the short programs in this section are doing.

Sections of Assembly Language are entered as part of a BASIC program, separated from the BASIC part by the square brackets [and]. The general structure of a program containing an assembler routine is:

```
1Ø REM BASIC program
1ØØ [
11Ø  \ Start of assembler mnemonics

2ØØ ]
21Ø REM BASIC program continues
```

Notice that remarks in the Assembly Language section are signalled by a backslash \ . The assembler then knows to ignore them.

Before the routine can be assembled, the computer must be told where it is to be put in computer memory. So the first line of the BASIC part must allocate some memory for this purpose, and there are two ways in which you can do this.

On entering the assembler routine, you assign to the resident integer variable **P%** the value you choose to be the address of the first instruction of the assembled machine code. **P%** is the 'pseudo program counter', used by the assembler, to calculate addresses for branch and jump instructions and as the pointer for the assembled codes. (When **O%** is not being used).

The two methods for doing this are:

(i) By direct assignment: **P% = &2ØØØ** for example. The problem with direct assignment is that you have to ensure that the memory location chosen is available for use.

The second method gets round this problem.

(ii) By using the BASIC **DIM** instruction. This takes the form **DIM P% 1ØØ**. Note the use of spaces, and no commas or brackets, to distinguish it from an array dimension. **DIM P% 1ØØ** allocates 1Ø1 bytes of memory for the machine-code, which will be stored along with all the BASIC variables above **LOMEM**. The number used with the **DIM** instruction must be large enough so that sufficient space is reserved to hold all the code, but not so large as to overlap other items in the memory.

An even better way in which to use **DIM** is: **DIM O% 1ØØ** followed by **P% =O%**. **DIM** is a convenient way of reserving space for machine code routines. No check is made to prevent the assembled code from overrunning the space reserved for it.

Assembly

To get the computer to assemble the routine into machine-code, you simply **RUN** the program. To complete the assembly, the program has to be **RUN** twice. The reason for this will become clear in a moment. The assembler pseudo-operator **OPT** controls the listing and error output generated on assembly. This operator must be placed in the assembler routine, usually at the start, and is followed by a number from Ø to 3 which causes the following outputs:

OPT0 No errors printed, no listing given.
OPT1 No errors, but a listing is given.
OPT2 Errors are printed, but no listing.
OPT3 Both errors and a listing are given.

The listing given is of the machine-code, in hexadecimal. The errors are printed as messages on the screen.

Here's an Assembly Language routine:

```
10 DIM O% 100
20 P% = O%
30 [OPT3
40 LDA &70
50 CMP #0
60 BEQ Zero
70 STA &72
80 .Zero RTS
90 ]
```

When you **RUN** this program, the computer will print a listing, and then the message:

```
No such variable at line 60
```

Routines which have forward reference to labels (Zero is referred to on line 60 when the assembler has not yet come across it) will always generate an error. The answer to this is to inhibit errors the first time through by using **OPT0**, and then to **RUN** a second time to generate the complete code. This is called two-pass assembly.

The way to do this is to enclose the routine in a **FOR**. . .**NEXT** loop as follows:

```
10 DIM O% 100
20 FOR I = 0 TO 3 STEP 3
30 P% = O%
40 [OPT I
50 LDA &70
60 CMP #0
70 BEQ Zero
```

```
8Ø STA &72
9Ø .Zero RTS
1ØØ ]
11Ø NEXT
```

On the first time through the loop, I = Ø and so there will be no listing and no error reported. This run allows the computer to identify the forward referenced label. The second time through the loop, I=3 and hence a list of compiled code is produced, along with any programming errors. Note that the assignment statement **P% = O%** is enclosed within the loop so that it is reset before each pass.

On running this program, you will see a listing of the assembled machine-code alongside the Assembly Language mnemonics:

```
>RUN
ØE75                    OPT I
ØE75 A5 7Ø              LDA &7Ø
ØE77 C9 ØØ              CMP #Ø
ØE79 FØ Ø2              BEQ Zero
ØE7B 85 72              STA &72
ØE7D 6Ø                 .Zero RTS
```

This means that the mnemonics have been successfully assembled, and the corresponding machine-code has been loaded into addresses &ØE75 to &ØE7D. &A5 is stored in location &ØE75, &7Ø in location &ØE76, &C9 in location &ØE77, and so on to 6Ø which is stored in location &ØE7D. This is nine bytes of machine-code in all.

This routine has not yet been executed. To do that a **CALL** from BASIC is required:

```
CALL O% RETURN
```

Nothing is printed on the screen when you do this, and that's because the program is trivial; it merely loads a byte from memory location &7Ø into the accumulator, and if it isn't zero it is stored in memory location &72. There are some points to note about the structure of the Assembly Language routine:

- When a label is assigned to a line, as at line 9Ø, it must be preceded by a full stop. When the label is called by an instruction, as at line 7Ø, there must be no full stop.
- Most Assembly Language routines end with **RTS** (return from subroutine) which transfers control back to the BASIC interpreter.
- The above routine uses two locations in the zero page of memory. Only locations &7Ø to &8F in the zero page may be used by your own programs; all the remainder is taken up by the operating system's variables, and BASIC's workspace.

Execution by USR

USR is similar to a BASIC **FN** (function); it gives a single value.

The format is:

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

where Z may be a label pointing to the first assembler mnemonic, or the address of the first instruction in machine-code. A label is easier to use since it requires no knowledge of where the machine-code is placed in memory. When **R% = USR(Z)** is executed, the least significant byte of each of the BASIC integer variables **A%**, **X%**, and **Y%** is placed into the accumulator, X register, and Y register respectively. The least significant bit of **C%** is placed in the carry flag (bit C of the flags register). **A%**, **X%**, **Y%**, and **C%** can therefore be used to initialise the 65Ø2 registers before entry into the assembler routine. Control then passes to the subroutine pointed to by Z. On returning to BASIC (after **RTS**), the four bytes comprising **R%** will each contain the contents of one of the 65Ø2 registers, as follows:

```
R% = PYXA
```

So **R%** contains the flags, Y register, X register, and accumulator in that order.

Any or each of these registers may be extracted from **R%** by setting up a *mask* using **AND**. To get the accumulator, the least significant byte is required:

```
Acc  = R% AND &FF
```

Similarly for X, Y:

```
X    = (R% AND &FF00) DIV &100
Y    = (R% AND &FF0000) DIV &10000
```

To get the flags:

```
10 DIM BLOCK 3
20 !BLOCK = USR(Z)
```

Then (Acc = **BLOCK?0**, X = **BLOCK?1**, Y = **BLOCK?2**), the flags = **BLOCK?3**

Here is a program which uses **USR**. The Assembly Language routine adds the numbers held in **X%** and **A%**, and gives the result in the accumulator:

```
10 DIM O% 100
20 FOR I = 0 TO 3 STEP 3
30 P% = O%
40 [OPT I
50 .Start STA &80
60 TXA
70 CLC
80 CLD
90 ADC &80
100 RTS
110 ]
120 NEXT
130 INPUT "First number "A%
140 INPUT "Second number "X%
150 Registers% = USR(Start)
160 Sum% = Registers% AND &FF
170 PRINT "Sum of two numbers is ";Sum%
```

When **RUN**, you will see the following:

```
>RUN
0F0A                OPT I
0F0A 85 80          .Start STA &80
0F0C 8A             TXA
0F0D 18             CLC
0F0E D8             CLD
0F0F 65 80          ADC &80
0F11 60             RTS
First number        11
```

Second number 12
Sum of two numbers is 23

The numbers 11 and 12 are entered by the user, and are stored in the integer variables **A%** and **X%**. The **USR** call tells the computer to start executing the assembly routine from the label Start. Before this happens, the least significant byte of **A%** is placed in the accumulator, and the least significant byte of **X%** into the X register. The machine-code corresponding to the assembler mnemonics is now executed in sequence:

STA &8Ø stores the contents of the accumulator in memory location &8Ø.

TXA transfers the contents of the X register to the accumulator.

CLC clears the carry flag prior to addition. If this is not done then a spurious carry may be added to give an incorrect result.

CLD clears the D flag so that the 65Ø2 is working in binary mode.

ADC &8Ø adds the contents of the accumulator to the contents of memory location &8Ø, plus the contents of the carry flag; and places the result in the acccumulator.

RTS returns control to BASIC.

Back in the BASIC section, **Registers%** now contains the four 65Ø2 registers' contents. The result is in the accumulator, so the least significant byte of **Registers%** is placed into **Sum%**, which is then printed to give the answer. Note that this routine performs only a single-byte addition, so any result given in **Sum%** will be MOD 256.

Execution by CALL

CALL is similar to a BASIC **PROC** (procedure).

Here is another addition routine:

```
1Ø DIM O% 1ØØ
2Ø FOR I=Ø TO 3 STEP 3
3Ø P% = O%
4Ø [OPT I
5Ø .Start CLC
6Ø CLD
7Ø LDA &8Ø
8Ø ADC &81
```

```
90 STA &82
100 RTS
110 ]
120 NEXT
130 INPUT "First number "number1%
140 INPUT "Second number "number2%
150 ?&80 = number1%
160 ?&81 = number2%
170 CALL Start
180 Sum% = ?&82
190 PRINT "Sum of two numbers is ";Sum%
```

This program illustrates the use of the indirection operator **?**. Indirection operators are very useful when calling assembly routines.

Here is a list to refresh your memory:

?&80 = J%	Will put the least significant byte of **J%** in location &80.
!&80 = &12345678	Will put &78 in location &80, &56 in location &81, &34 in location &82, and &12 in location &83.
$V% = "FAULT"	Will put the string **"FAULT"** plus a carriage return (**¦M**) in locations starting at **V%**. **V%** must not be in zero page.
S% = ?&80	Will read the contents of location &80 (1 byte) into **S%**.
R% = !&87	Will read 4 bytes from locations &87 to &8A into **R%**; &8B being the most significant, &87 the least significant.
R$ = $&2000	Will read a string starting at &2000 into **R$**.

The addition program shown above has exactly the same effect as the previous example. In this instance though, the two numbers are stored into memory in the BASIC part of the program, and are added and the result stored in the Assembly Language part.

CALL may also be used with parameters, similar to **PROC**. This takes the form:

```
CALL Start,integer%,decimal,string%,?byte
```

The parameters are separated by commas. **Start** is a label, but could equally well be a specific address, &2000 for example. The above **CALL**

shows that any kind of variable may be passed as a parameter: integer, real, string, and single-byte. When a **CALL** is made, the parameters are assigned to a *parameter block,* which starts at memory location &6ØØ. The format of this parameter block is:

Address	Contents
&6ØØ	Number of parameters
&6Ø1	1st parameter address (low)
&6Ø2	1st parameter address (high)
&6Ø3	1st parameter type
&6Ø4	2nd parameter address (low)
&6Ø5	2nd parameter address (high)
&6Ø6	2nd parameter type

There may be any number of parameters, and this number is given in the first byte of the parameter block. Following this, each parameter's address and type is given.

The type is designated by a number:

Ø	A single byte (eg ?location)
4	A 4-byte variable (eg Z% or !address)
5	A 5-byte variable (eg number)
128	A defined string (eg "YES PLEASE") which must end with &D (**RETURN**)
129	A string variable (eg name$)

The way that the parameter block is laid out, it would seem that the best way to access the individual parameters is to use indirect addressing. Unfortunately, the 65Ø2 only allows the zero-page to be used for indirect address pointers, so here is a routine which transfers the addresses from the parameter block into free locations in the zero-page:

```
        LDA &6ØØ      \Check the number of parameters.
        BEQ End        \If zero then finish.
        STA &7Ø        \If not then store this number.
        LDX #Ø         \Clear the X register
        LDY #Ø         \and the Y register.
.Loop   LDA &6Ø1,Y     \Take high address of parameter.
        STA &71,X      \and store it in zero-page.
        INX            \Increment X register.
```

```
        INY             \ and Y register.
        LDA &6@1,Y      \ Take low address of parameter.
        STA &71,X       \ and store it in zero-page.
        INX             \ Increment X register
        INY             \ and Y register
        INY             \ twice.
        DEC &7@         \ Decrement number of parameters.
        BNE Loop        \ If still not zero then repeat.
.End    RTS             \ Return to BASIC.
```

This routine stores the address of each parameter in zero-page memory starting at location &71. 15 parameter addresses may be stored in this way before the total user zero-page memory is filled. This routine is very useful if the number of parameters passed to a particular Assembly Language subroutine is not always the same, for it will only relocate the addresses of those parameters which exist.

Here this routine is incorporated into another addition program:

```
1@ DIM Q% 1@@
2@ FOR I=@ TO 3 STEP 3
3@ P% = Q%
4@ [OPT I
5@ .Start CLC
6@ CLD
7@ LDA &6@@
8@ BEQ End
9@ STA &7@
1@@ LDX #@
11@ LDY #@
12@ .Loop1 LDA &6@1,Y
13@ STA &71,X
14@ INX
15@ INY
16@ LDA &6@1,Y
17@ STA &71,X
18@ INX
19@ INY
2@@ INY
21@ DEC &7@
22@ BNE Loop1
```

```
230 .End LDX #0
240 STX &2000
250 LDY &600
260 BEQ Finish
270 .Loop2 LDA (&71,X)
280 ADC &2000
290 STA &2000
300 INX
310 INX
320 DEY
330 BNE Loop2
340 .Finish RTS
350 ]
360 NEXT
370 INPUT"First number "one%
380 INPUT"Second number "two%
390 INPUT"Third number "three%
400 CALL Start,one%
410 Sum%=?&2000
420 PRINT Sum%
430 CALL Start,one%,two%
440 Sum%=?&2000
450 PRINT Sum%
460 CALL Start,one%,two%,three%
470 Sum%=?&2000
480 PRINT Sum%
```

The parameter block transfer routine ends at line 240, where the addition routine begins. Notice that the whole routine is **CALL**ed with varying numbers of parameters, just to prove that it works. The result of adding the parameters is given in location &2000. However, as with the previous programs, the result is MOD 256.

Quadruple precision addition

Integer variables are stored in four consecutive bytes of memory. Groups of four bytes can be accessed using !, and can be added together. This is achieved a byte at a time, starting with the least significant, and storing each successive result:

```
10 DIM Q% 100
```

```
20 FOR I = 0 TO 3 STEP 3
30 P% = O%
40 [OPT I
50 .Start CLC                    \Clear carry for ADC instruction
60 CLD
70 LDX #0                        \ Clear X register
80 LDY #4                        \ Set Y register to 4 as a counter
90 .Loop LDA &70,X               \ Put byte from one% in accumulator
100 ADC &74,X                    \ Add byte from two%
110 STA &78,X                    \ Store result
120 INX                          \ Increment X register
130 DEY                          \ Decrement Y register
140 BNE Loop                     \ If not zero then repeat
150 RTS
160 ]
170 NEXT
180 INPUT "First number "one%
190 INPUT "Second number "two%
200 !&70 = one%
210 !&74 = two%
220 CALL Start
230 sum% = !&78
240 PRINT "Sum of two numbers is ";Sum%
```

This program will work with positive or negative integers.

Multiplication

The 6502 does not have a multiply instruction. Multiplication is achieved by adding and shifting, just like ordinary decimal long-multiplication. As a simple example, take the multiplication of two 4-bit numbers. Such a multiplication can give an 8-bit result:

(i) Test the rightmost bit of the multiplier. If it is zero then add 0000 to the most significant end of the result. If it is 1 then add the number to be multiplied to the most significant end of the result.

(ii) Shift the result one bit position to the right. Repeat (i) for the next bit of the multiplier.

Applying the above to 1101*1001, the rightmost bit of the multiplier (1001) is 1. Therefore 1101 is added to the most significant end of the

result:

11Ø1

Shift the result right one bit position:

Ø11Ø1

The next bit of the multiplier is zero, so ØØØØ is added to the result, and it is again shifted right:

ØØ11Ø1

The next bit is again zero:

ØØØ11Ø1

The final bit is 1, so 11Ø1 is added to the result, and the final shift is performed:

Ø111Ø1Ø1

Notice that for 4-bit multiplication, four shifts are required. 8-bit multiplication will require eight shifts, 16-bit multiplication 16 shifts, and so on.

To put the above routine into practice on the 65Ø2, the shift and rotate instructions are used. Here is a program to multiply two 8-bit numbers:

```
10 DIM O% 100
20 FOR I=0 TO 3 STEP 3
30 P% = O%
40 [OPT I
50 .Start CLD
60 LDA #0
70 STA &72          \ Clear 16-bits
80 STA &73          \ for the result.
90 LDY #8           \ Set Y to 8 as a counter.
100 .Loop LSR &71   \ Shift multiplier right one bit.
110 BCC NoAdd       \ Test this bit. Branch if zero.
120 CLC             \ Clear carry prior to addition.
```

```
13Ø LDA &7Ø                 \ Load accumulator with number to be
                            \ multiplied.
14Ø ADC &73                 \ Add most significant byte of result.
15Ø STA &73                 \ Store in most significant byte of result.
16Ø .NoAdd ROR &73          \ Shift result right, with carry
17Ø ROR &72                 \ from addition.
18Ø DEY                     \ Decrement counter.
19Ø BNE Loop                \ Repeat if not zero.
2ØØ RTS
21Ø ]
22Ø NEXT
23Ø INPUT "First number "one%
24Ø INPUT "Second number "two%
25Ø ?&7Ø = one%
26Ø ?&71 = two%
27Ø CALL Start
28Ø Product% = ?&72 + 256*?&73
29Ø PRINT "Product of two numbers is ";Product%
```

This routine is not the most efficient way of multiplying two bytes together, but it illustrates the method clearly:

Lines 6Ø, 7Ø and 8Ø clear the two bytes in memory which will be used for the result of the multiplication. These locations are &72 (result low byte) and &73 (result high byte).

Lines 25Ø and 26Ø store the numbers to be multiplied in locations &7Ø and &71. It doesn't matter which of these is chosen to be the multiplier; the example uses the number in &71.

Line 9Ø sets the Y register to 8 as a counter. Because this is an 8-bit multiplication, eight shifts are required.

Line 1ØØ shifts the multiplier right one bit position. The rightmost bit falls into the carry where it can be tested.

Line 11Ø carries out the test. If the C bit is zero then the program branches to **NoAdd**; if it is 1 then the addition of the number in &7Ø to the result high byte (&73) takes place.

Lines 12Ø to 15Ø accomplish this addition, by clearing the carry bit, loading the accumulator from &7Ø, adding the result high byte, and then storing back in the result high byte.

Line 16Ø, labelled **NoAdd**, rotates the result high byte right one bit

position. The carry from the addition in line 14Ø is entered from the left, and the rightmost bit falls into the carry.

Line 17Ø rotates the result low byte right one bit position. The leftmost bit from the high byte, now in the carry, enters the low byte from the left.

Line 19Ø decrements the counter, and repeats the above process until the counter is zero.

The program will give the result of multiplying two positive integers, each between Ø and 255. You can see how many instructions it takes just to do this, and can imagine the complexity of a BASIC statement when it is interpreted into machine-code.

A shorter routine to multiply two bytes uses the accumulator as the result high byte, and the multiplier as the result low byte. As each bit of the multiplier is shifted into the carry to be tested, the leftmost bit of the multiplier location becomes vacant, so allowing the result to be shifted in.

```
.Start      CLD
            LDA #Ø          \ Clear result high byte.
            LDY #8          \ Set shift counter.
.Loop       ROR &71         \ Shift multiplier right one bit.
            BCC NoAdd       \ Test this bit. Branch if zero.
            CLC             \ Clear carry prior to addition.
            ADC &7Ø         \ Add number to be multiplied.
.NoAdd      ROR A           \ Shift result right, with carry from addition.
            DEY             \ Decrement counter.
            BNE Loop        \ Repeat if not zero.
            ROR &71         \ Final shift of result.
            STA &72         \ Store result high byte.
            RTS
```

Before using this routine, the two bytes to be multiplied are placed in locations &7Ø and &71. The result appears in &71 (low byte) and &72 (high byte).

To multiply two 4-byte numbers together, the additions and shifts must act on each byte in turn, and the total number of shifts must be 32.

```
.Start      CLD
            LDX #8          \ Clear
            LDA #Ø          \ eight
```

```
.Clear      STA &77,X     \ bytes
            DEX           \ for
            BNE Clear     \ result.
            LDY #32       \ Set shift counter.
.Loop       LSR &77       \ Shift four bytes
            ROR &76       \ of multiplier
            ROR &75       \ right
            ROR &74       \ one bit.
            BCC NoAdd     \ Test this bit. Branch if zero.
            CLC           \ Clear carry prior to addition.
            LDA &70       \ Add
            ADC &7C       \ 4-byte
            STA &7C       \ multiplier
            LDA &71       \ to
            ADC &7D       \ 4-byte
            STA &7D       \ result
            LDA &72       \ and
            ADC &7E       \ store.
            STA &7E       \ "
            LDA &73       \ "
            ADC &7F       \ "
            STA &7F       \ "
.NoAdd      LDX #8        \ Shift
.Shift      ROR &77,X     \ eight bytes
            DEX           \ of result
            BNE Shift     \ right
            DEY           \ one bit.
            BNE Loop      \ Repeat if not zero
            RTS
```

Before using this routine, the two numbers to be multiplied must be placed in **!&70** and **!&74**. The result appears in the four bytes from &78 (least significant) to &7B (most significant), and is accessed as **!&78**. This routine will work with both positive and negative integers.

Division

Division is accomplished as the reverse of multiplication. 8-bit multiplication gave a 16-bit result, so, for division, a 16-bit numerator and 8-bit denominator will give an 8-bit result. The numerator is stored in two bytes of memory. It is shifted left one bit position and the

numerator high byte is then loaded into the accumulator. If the shift produced a carry then a 1 is shifted left into the result, the denominator is subtracted from the accumulator, and the accumulator contents are then stored in the numerator high byte. If the shift did not produce a carry then the denominator is subtracted from the accumulator in any case. If this subtraction produces a carry then a 1 is shifted left into the result and the accumulator contents are stored in the numerator high byte. If no carry, then Ø is shifted left into the result.

This whole process is repeated eight times. The division program is as follows:

```
10 DIM Q% 100
20 FOR I=0 TO 3 STEP 3
30 P% = Q%
40 [OPT I
50 .Start CLD
60 LDY #8              \ Set shift counter.
70 .Loop ASL &72       \ Shift numerator
80 ROL &73             \ left one bit.
90 LDA &73             \ Load accumulator high byte.
100 BCC Label          \ Test carry produced by shift.
110 SBC &71            \ Subtract denominator and
120 STA &73            \ store in numerator high byte.
130 SEC                \ Set carry prior to shifting into result.
140 JMP Shift          \ Go to Shift.
150 .Label SEC         \ Set carry prior to subtraction.
160 SBC &71            \ Subtract denominator
170 BCC Shift          \ and test carry.
180 STA &73            \ Store in numerator high byte.
190 .Shift ROL &70     \ Shift either 0 or 1 into result.
200 DEY                \ Decrement counter.
210 BNE Loop           \ Repeat if not zero.
220 RTS
230 ]
240 NEXT
250 INPUT "Numerator "numerator%
260 INPUT "Denominator "denominator%
270 P% = &71
280 [OPT 3
290 EQUB denominator%  \ Store denominator at location &71
300 EQUW numerator%    \ Store numerator at locations &72 and &73
```

```
310 RTS
320 ]
330 CALL Start
340 PRINT "Quotient is ";?&70
350 PRINT "Remainder is ";?&73
```

In this routine, the denominator is stored at location &71, and the numerator in two bytes &72 and &73. The result appears in &70, and any remainder is left in &73. Remember, this is a 16-bit by 8-bit division, so the denominator may not be greater than 255 and the numerator not greater than 65025 (255^2) to give a valid result (the result must be 255 or less).

The short routine from lines 280 to 320 is used to store the data in memory, and contains some instructions which you have not yet seen or used. **EQUB** and **EQUW** are in the same class of instruction as **OPT**, in that they are used in the Assembly Language part of the program but are not assembly instructions. They are used simply to store data at the location(s) at which they appear when assembled into machine-code. You will see this clearly when you **RUN** the above program. After you have typed in the numerator and denominator you will see a listing of the machine-code from &0071 to &0074.

There are in fact four **EQU** instructions:

EQUB stores a byte of data.
EQUW stores a word of data (2 bytes).
EQUD stores a double-word of data (4 bytes).
EQUS stores the ASCII representation of a string.

EQUS is illustrated in the next section on error handling in Assembly Language.

Notice in the program example above how putting P% equal to &71 enables the denominator to be stored in &71 using **EQUB**, and the numerator to be stored in &72 and &73 using **EQUW**. **EQUD** may be used to store the contents of a full BASIC integer variable. (You may use **EQUB** instead of **?**, and **EQUD** instead of **!**.)

Error trapping in assembler

The assembler will tell you of any mistakes which you make in typing in programs (syntax errors), and some errors associated with BASIC variables during assembly, but there is no such thing as a run-time error

in machine-code: you just have to fathom it out line by line. However, it is possible for you to trap errors generated while a machine-code program is running by using the **BRK** instruction. As an example, take the division program described in the previous section. Everyone knows that it is not possible to divide by zero, but the program does not know this. If you try to do so it unwittingly gives the answer 255.

It is simple to test the denominator before the division is started, and then to branch to an error routine. The whole program is not repeated here, but the following lines may be added:

```
53 LDA &71
56 BEQ Error

222 .Error BRK
224 EQUB 18
226 EQUS "Division by zero"
228 BRK
```

If you now run the program with a zero denominator, it will stop and print the message:

```
Division by zero at line 330
```

You can also type

```
PRINT ERR RETURN
```

upon which it will give the correct error number, 18.

Any error message must take the following form:

```
BRK
EQUB errornumber (ERR)
EQUS "message"
BRK
```

Operating system calls from assembler

All the operating system calls available from BASIC, and many more, are available from a machine-code program. These routines are always accessed using a **JSR** to some address in the operating system, and usually involve the passing of one or more parameters via the accumulator (for 1), X and Y (for 2 or 3), or a parameter block in memory

(for more than 3).

Here is a table showing all the operating system calls available.

Routine		Vector		Summary of function
Name	Address	Name	Address	
		UPTV	222	User print routine
		EVNTV	22Ø	Event interrupt
		FSCV	21E	File system control entry
OSFIND	FFCE	FINDV	21C	Open or close 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	FFEO	RDCHV	21Ø	Read character (from keyboard) to A
OSASCI	FFE3	–	–	Write a character (to screen) from A plus LF if (A)=&ØD
OSNEWL	FFE7	–	–	Write LF,CR (&ØA, &ØD) to screen
OSWRCH	FFEE	WRCHV	2ØE	Write character (to screen) from A
OSWORD	FFF1	WORDV	2ØC	Perform miscellaneous OS operation using control block to pass parameters
OSBYTE	FFF4	BYTEV	2ØA	Perform miscellaneous OS operation using registers to pass parameters
OSCLI	FFF7	CLIV	2Ø8	Interpret the command line given

When you use one of these routines, you must use a **JSR** to the corresponding address shown in the second column. For example, OSWRCH is called from assembler by typing:

```
JSR &FFEE
```

The routine stored at **&FFEE** uses the OSWRCH vector address, shown in the fourth column, as an indirect pointer to the actual location of the OSWRCH routine.

The reason for this is twofold:

(i) The actual address of the OSWRCH routine may be altered by the manufacturer without affecting the operating system subroutine call in any way. **JSR &FFEE** will always give an OSWRCH call even though the address held in locations &20E and &20F may not be the same on every machine.

(ii) The user can alter the address held in the zero-page vector location and trap any call of that particular operating system routine, indirecting such a call to the user's own routine anywhere in memory.

Use of operating system calls

OSWRCH entry: &FFEE vector: &20E
This call writes the character whose ASCII code is in the accumulator to the screen.

Here is an example which will print the character L on the screen:

```
10 P% = &700
20 [OPT 3
30 .Start LDA #76     \ Load accumulator with ASCII code for L
40 JSR &FFEE          \ Jump to.OSWRCH
50 RTS
60 ]
70 CALL Start
```

OSWRCH is also used with ASCII control codes (from 0 to 31). If you change line 30 to:

```
30 .Start LDA #7
```

then the program will output ASCII character 7, which is a 'beep'.

Some BASIC instructions have ASCII values in the control code range, and these can therefore be used with OSWRCH. For example, **PLOT** has an ASCII value of 25, **TAB** an ASCII value of 31.

The following program uses **TAB** to print the character L half way across the screen:

```
10 P% = &70
20 [OPT 2
30 .Start LDA #31
40 JSR &FFEE
50 LDA #19
60 JSR &FFEE
70 LDA #VPOS
80 JSR &FFEE
90 LDA #76
100 JSR &FFEE
110 RTS
120 ]
130 CALL Start
```

Each parameter is passed in turn to OSWRCH via the accumulator. The BASIC statement equivalent to the above program is:

```
PRINT TAB(19);"L";
```

(Note that this program will not work with OPT 3 because **VPOS** is affected.)

The BASIC instruction **PLOT** takes three parameters, **PLOT A,X,Y**. However, X may be 0 to 1279 and Y may be 0 to 1023, so each must be represented by two bytes. That means that an OSWRCH call with the accumulator set to 25 must be followed by five more OSWRCH calls to pass the parameters. The following program will plot a line on the screen:

```
10 MODE 4
20 P% = &70
30 [OPT 3
40 .Start LDA #25
50 JSR &FFEE
```

```
60 LDA #5
70 JSR &FFEE
80 LDA #88
90 JSR &FFEE
100 LDA #2
110 JSR &FFEE
120 LDA #44
130 JSR &FFEE
140 LDA #1
150 JSR &FFEE
160 RTS
170 ]
180 CALL Start
```

This program is equivalent to

```
PLOT5,600,300
```

Lines 100 and 80 give X (2*256 + 88) and lines 140 and 120 give Y (1*256 + 44).

You'll see from the listing that the above routine, when assembled, occupies memory from &70 to &8E. Remember that user programs must use zero-page locations only between &70 and &8F, so this is almost the largest size routine that may be stored in the zero-page.

OSASCI entry: &FFE3
Writes the character whose code is in the accumulator to the screen using OSWRCH. However, if the accumulator contains &D then OSNEWL is called instead. The actual code at location &FFE3 is:

```
.OSASCI     CMP #&D
            BNE OSWRCH
.OSNEWL     LDA #&A
            JSR OSWRCH
            LDA #&D
.OSWRCH     JMP (WRCHV)
```

OSNEWL entry: &FFE7
This call issues a line feed/carriage return to the screen, as shown above.

After using OSWRCH, OSASCI, or OSNEWL, the contents of the accumulator, X, and Y registers are unchanged. Flags C, N, V, and Z are undefined, and D = Ø.

OSRDCH entry: &FFEØ vector: &21Ø
This call reads a character code from the keyboard into the accumulator.

After using OSRDCH, the contents of the X and Y registers are unchanged. Flags N, V, and Z are undefined, and D = Ø. Flag C tells whether the read has been successful (C = Ø). If C = 1 then an error has occurred and the error number is given in the accumulator. If C = 1 and A = &1B then an escape condition has been detected and you must acknowledge this by performing an OSBYTE call with A = &7E, or ***FX126**.

OSCLI entry: &FFF7 vector: &2Ø8
This call is used by the BASIC OSCLI instruction. From assembler it consists of a **JSR** to &FFF7, the command line string being placed in memory at a location given by the contents of the X register (address low byte) and Y register (address high byte). The command line string must be terminated by &D **RETURN**.

The following BASIC program illustrates this:

```
1Ø DIM address 2Ø
2Ø keynumber = 4
3Ø $address = "KEY"+STR$ keynumber+"LIST¦ M"
4Ø X% = address MOD 256
5Ø Y% = address DIV 256
6Ø CALL &FFF7
```

This will have the same effect as

```
*KEY4"LIST¦M"
```

Note: The string indirection operator **$** automatically puts a **RETURN** code (&D) after the string. **EQUS** however does not, and it must be inserted afterwards using **EQUB &D** or something like **EQUS "FRED" + CHR$13**.

OSFIND entry: &FFCE vector: &21C
Opens a file from cassette or disc for reading or writing. The contents of the accumulator determine the operation performed:

A = 0	closes a file or files (**CLOSE#**).
A = &40	opens a file for input (**OPENIN**).
A = &80	opens a file for output (**OPENOUT**).
A = &C0	opens a file for input or output (**OPENUP**).

When **OPENIN**, **OPENOUT**, or **OPENUP** is used, X and Y must contain the address of the filename. After the subroutine call, the accumulator will contain the channel number allocated to that file by the operating system.

If **CLOSE#** is used then Y must contain the channel number of the file to be closed. If Y is 0 then all files will be closed.

OSBPUT entry: &FFD4 vector: &218
Writes the byte contained in the accumulator to the cassette or disc file (same as **BPUT#**). Y must contain the file channel number. After using **OSBPUT**, the contents of the accumulator, X, and Y registers are unchanged.

OSBGET entry: &FFD7 vector: &216
Reads a byte from the cassette or disc file into the accumulator (same as **BGET#**). Y must contain the file channel number. After using **OSBGET**, the contents of the X and Y registers are unchanged. Flags N, V, and Z are undefined, and D = 0. Flag C tells whether the read has been successful (C = 0). If C = 1 then an error has occurred and the error number is given in the accumulator. If C = 1 and A = &FE then the end of file has been reached.

OSFILE entry: &FFDD vector: &212
Allows a whole file to be loaded or saved. The contents of the accumulator indicate the function to be performed. X and Y point to an 18 byte control block anywhere in memory, the structure of which is as follows:

OSFILE control block

00	Address of file name, which must be terminated	LSB
01	by &0D	MSB

Ø2	Load address of file	LSB
Ø3		
Ø4		
Ø5		MSB
Ø6	Execution address of file	LSB
Ø7		
Ø8		
Ø9		MSB
ØA	Start address of data for write operations, or length of file for read operations	LSB
ØB		
ØC		
ØD		MSB
ØE	End address of data, that is byte after last byte to be written or file attributes	LSB
ØF		
1Ø		
11		MSB

The table below indicates the function performed by OSFILE for each value of A.

A=Ø	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.

Note: Values 1 to 6 are not available on a cassette filing system.

OSBYTE entry: &FFF4 vector: &2ØA
This is a family of operating system calls which includes all the ***FX** calls available from BASIC. (These are not repeated here.) The call number is passed in the accumulator and parameters are passed in X or Y or both.

All OSBYTE calls are available from BASIC via a **USR** call, or by using a ***FX** call.

Here is a list of functions as given by each accumulator value (A):

A = 127 (**EOF#**) ***FX127**

Gives the end of file status of a previously opened file. X must contain the file channel number. Afterwards, X will be zero if the end of file has not been reached, non-zero if the end of file has been reached.

A = 129 (**INKEY**) ***FX129**

Either waits for a character from the keyboard buffer until a time limit expires (**INKEY**positive) or tests if a key is depressed (**INKEY**negative). All the discussion about auto-repeat and buffer flushing applies to this call.

For **INKEY**positive, Y must contain the most significant byte of the delay, and X the least significant (in hundredths of a second).

Afterwards, if Y = Ø then a character has been detected and its code appears in X. Y = &1B indicates that **ESCAPE** was pressed and must be acknowledged with ***FX126**. Y = &FF indicates that no key was pressed in the allocated time.

For **INKEY**negative, Y must contain the requisite key-code in twos complement. Afterwards Y will be either **TRUE** (&FF) or **FALSE** (zero) depending on whether the key was pressed.

A = 131 (OSHWM) ***FX131**

Gives the address of the first free location in memory above that required for the operating system. Usually equal to &EØØ. The address is given in X (low byte) and Y (high byte). For example, after ***FX2Ø,6**.

A = 132 ***FX132**

Gives the lowest memory address used by the screen display in X (low byte) and Y (high byte).

A = 133 Low mode address ***FX133**

Gives the lowest address in memory used by a particular mode. Does not change mode but merely investigates the consequences of doing so. The

mode to be investigated must be in X. Afterwards, the address is contained in X (low byte) and Y (high byte).

A = 134 Read position of text cursor ***FX134**

Gives in X the X coordinate of the text cursor, and in Y the Y coordinate (same as **POS** and **VPOS**).

A = 135 Read character at position of text cursor ***FX135**

Gives in X the ASCII code of the character at the current text cursor position, and in Y the current mode number. X is Ø if the character is not recognisable.

Here is a BASIC function which can be used to read the character at any position X,Y on the screen:

```
1ØØØ DEF FNreadcharacter(column%,row%)
11ØØ LOCAL A%,currentX%,currentY%,character%
12ØØ currentX% = POS: currentY% = VPOS
13ØØ VDU31,column%,row%
14ØØ A% = 135
15ØØ character% = (USR(&FFF4) AND &FFØØ) DIV &1ØØ
17ØØ VDU31,currentX%,currentY%
18ØØ = CHR$ character%
```

To give the character at position X,Y this function would be called by passing X and Y as the two parameters:

```
PRINT FNreadcharacter(X,Y)
```

A = 137 Motor control ***FX137**

Similar to ***MOTOR**. X = Ø will turn off the cassette motor, X = 1 will turn it on.

A = 139 (***OPT**) ***FX138**

Exactly the same as ***OPT**. The parameters are passed in X and Y.

A = 145 Get character from keyboard buffer ***FX145**

Reads a character code from a buffer into the Y register. X = buffer number (Ø to 9 inclusive). C = Ø indicates a successful read, C = 1 indicates that the buffer is empty.

A = 218 Cancel **VDU** queue ***FX218**

Many **VDU** codes expect a sequence of bytes (as shown earlier with **PLOT** and **TAB**). This call signals the **VDU** software to throw away the bytes it has received so far. Before use, X and Y must contain zero.

OSWORD entry: &FFF1 vector: &20C

This is a family of operating system calls which uses a parameter block somewhere in memory to supply data to the routine and to receive results from it. The exact location of the parameter block must be specified in X (low byte) and Y (high byte). The accumulator contents determine the action of the OSWORD call.

A = Ø Read a line from keyboard to memory

Accepts characters from the keyboard and places them at a specified location in memory. During input the **DELETE** key (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 the **ESCAPE** key is pressed.

The control block contains five bytes:

YX	(low byte) Address at which
YX+1	(high byte) line is to be stored
YX+2	Maximum length of line
YX+3	Minimum acceptable ASCII value
YX+4	Maximum acceptable ASCII value

Characters will only be entered if they are in the range specified by YX+3 and YX+4.

Afterwards, C = Ø indicates that the line was terminated by a **RETURN**. C not equal to zero indicates that the line was terminated by an **ESCAPE**. Y is set to the length of the line, excluding the carriage return if C = Ø.

A = 1 Read clock

Reads the internal elapsed-time clock into the five bytes pointed to by X and Y. The clock is incremented every hundredth of a second, and is used by the BASIC variable **TIME**.

A = 2 Write clock

Sets the internal elapsed-time clock to the value given in the five bytes pointed to by X and Y. Location YX is the least significant byte of the clock, YX+4 is the most significant.

A = 3 Read interval timer

In addition to the clock there is an interval timer which is also incremented every hundredth of a second. The interval is stored in five bytes pointed to by X and Y. See OSWORD with A = 1.

A = 4 Write interval timer

X and Y point to five bytes which contain the new value to which the clock is to be set. The interval timer may cause an event when it reaches zero. Thus setting the timer to &FFFFFFFFFD would cause an event after three hundredths of a second.

A = 7 **SOUND**

Equivalent to the BASIC **SOUND** statement. The eight bytes pointed to by X and Y contain the four two-byte parameters (in fact only the least significant byte of each need be used).

YX	Q (channel, Ø to 3)
YX+1	zero
YX+2	A (envelope, –15 to 4)
YX+3	zero, or &FF if –1 or some other negative value
YX+4	P (pitch, Ø to 255)
YX+5	zero
YX+6	D (duration, 1 to 255)
YX+7	zero

A = 8 **ENVELOPE**

Equivalent to the BASIC **ENVELOPE** statement. X and Y point to 14 bytes of data which are the 14 parameters used by **ENVELOPE**.

A = 9 **POINT**

Equivalent to BASIC **POINT** function. The parameter block pointed to by X and Y must be set up as follows:

YX X (low byte) coordinate
YX+1 X (high byte) coordinate
YX+2 Y (low byte) coordinate
YX+3 Y (high byte) coordinate

Afterwards, YX+4 will contain the logical colour value of that particular graphics coordinate. If the coordinate is off the screen then YX+4 contains &FF.

A = 10 Read character definition

Characters are displayed on the screen as an 8×8 matrix of dots. The pattern of dots for each character, including user-defined characters, is stored as eight bytes. This call enables the eight bytes to be read into a block of memory starting at the address given in X and Y, plus 1. The ASCII code of the character must be the first entry on the parameter block when the routine is called.

Afterwards, the parameter block contains data as shown below:

YX Character code
YX+1 Top row of displayed character
YX+2 Second row of displayed character
.
.
.
YX+8 Bottom row of displayed character

Here is a program to illustrate this OSWORD call, and the method of calling OSWORDs in general. It takes each of the characters in turn, reads the matrix definition, and then reverses this definition by shifting the bits in each byte, and then redefining each character using **VDU23**. The result makes your program interesting to read, to say the least!

```
10 MODE 6
20 DIM Q% 100
30 FOR I=0 TO 3 STEP 3
40 P% = Q%
50 [OPT I
60 .Character LDA &6001  \ Take low address of parameter
70 STA &70               \ and store it in zero-page.
```

```
80 LDA &6002          \ Take high address of parameter
90 STA &71            \ and store it in zero-page.
100 LDY #0            \ Clear Y register.
110 LDA (&70),Y       \ Get parameter (ASCII code)
120 STA &70           \ and store it in zero-page.
130 LDX #&70          \ Set X to OSWORD parameter block address low
                      \ byte.
140 LDA #10           \ Set OSWORD function.
150 JSR &FFF1         \ Jump to OSWORD.
160 LDX #0            \ Clear X register.
170 .Reverse LDY #8   \ Set Y to 8 as shift counter.
180 .Loop ASL &71,X   \ Shift each byte into the byte
190 ROR &70,X         \ below, thereby reversing it.
200 DEY               \ Decrement Y.
210 BNE Loop          \ Repeat if not zero.
220 INX               \ Increment X.
230 CPX #8            \ Compare with 8.
240 BNE Reverse       \ Repeat if not equal.
250 RTS
260 ]
270 NEXT
280 *FX20,6
290 FOR I%=33 TO 126
300 PRINT CHR$ I%;
310 CALL Character,I%
320 VDU23,I%,?&77,?&76,?&75,?&74,?&73,?&72,?&71,?&70
330 NEXT
```

This program illustrates a number of the features demonstrated in this part of the book. It calls the machine-code routine *Character* with a parameter, and lines 60 to 120 transfer the parameter to location &70, which is a safe place at which to store any OSWORD data.

Line 130 sets X to the low byte of the address of the OSWORD parameter block (it is not necessary to set Y because Y is already zero).

Lines 140 and 150 carry out the OSWORD call.

Lines 160 to 240 reverse each of the bytes of the character definition.

Line 280 explodes the character memory allocation to its maximum, allowing all the characters to be redefined, and line 320 carries out this redefinition.

It should be noted that if this program were more than &3ØØ bytes long, it would get overwritten by the soft characters.

A = 11 Read colour assigned to logical value

Gives the actual colour value assigned to the logical colour value contained in the location pointed to by X and Y. Afterwards, location YX will contain the logical value, and location YX+1 will contain the actual value. In fact YX+1 to YX+4 contain the four-byte physical colour – you must reserve space for five bytes.

Events

Events are conditions which occur within the computer and which can be trapped by the user so as to provide useful information. For example, it is possible to detect when the keyboard buffer is full, or when **ESCAPE** has been pressed.

To be able to act upon an event, that event must first be enabled by ***FX14**:

***FX14,Ø** enables output buffer empty event.
***FX14,1** enables input buffer full event.
***FX14,2** enables character entering keyboard buffer event.
***FX14,4** enables start of vertical synchronisation of screen display event.
***FX14,5** enables the interval timer crossing zero event.
***FX14,6** enables **ESCAPE** pressed event.

The operating system detects all the above events when they occur, but ignores them if they have not been enabled with the appropriate ***FX14** call. If an event occurs which has been enabled then program execution indirects via &22Ø and places an event code (shown below) in the accumulator. The contents of X and Y may also depend upon the event.

The event codes are as follows:

A = Ø	Output buffer empty. (X contains buffer identity.)
A = 1	Input buffer full. (X contains buffer identity, Y contains the ASCII code of character that could not be stored in buffer.)
A = 2	Key pressed.
A = 3	Vertical synchronisation of screen display.
A = 5	Interval timer crossing zero.
A = 6	**ESCAPE** detected.

Any address may be stored in the two bytes &22Ø and &221 to which the program will transfer execution on detection of an enabled event. You may write your own code at this address in order to process the event, but it must be terminated by **RTS**, and should not take too long (one millisecond maximum).

Each of the events may be disabled by a corresponding ***FX13**. For example, ***FX13,1** will disable the input buffer full event.

Assembly Language mnemonics

This section describes, in alphabetical order, all the 65Ø2 assembler mnemonics.

The following abbreviations are used:

A	accumulator
X	index register X
Y	index register Y
F	flags register
PC	program counter
PCH	program counter (high byte)
PCL	program counter (low byte)
SP	stack pointer
M	memory address
←	'becomes' (assignment)
→	'affects' (flags)
( )	contents of
&	hexadecimal
(A4) (M7) (XØ)	etc specified bit position in register or memory
N V B D I Z C	status flags

Because this section has been written for use with the Electron's assembler, the addressing modes quoted are simplified as compared with those that are specified for use with general machine-code programming of the 65Ø2.

ADC — Add with carry

Action	A ← (A)+data+C
Description	Add the contents of memory location or immediate data to the accumulator, plus the carry bit. Result is placed in the accumulator.
Addressing modes	Immediate Zero-page Absolute Indexed Indirect/indexed
Flags affected	NVZC
Comments	To add without the carry, bit C must be cleared beforehand by using CLC.

AND — Logical AND

Action	A ← (A) AND data
Description	AND the contents of memory location or immediate data with the accumulator. Result is placed in the accumulator.
Addressing modes	Immediate Zero-page Absolute Indexed Indirect/indexed
Flags affected	NZ
Comments	See BASIC AND for truth table.

ASL — Arithmetic shift left

Action	C ← A or M ← ∅

Description	Shift the contents of the accumulator or memory location left one bit position. Bit 7 falls into the carry (bit C), zero is entered from the right. Result remains in either the accumulator or the memory location.
Addressing modes	Zero-page Absolute Indexed (X only) Accumulator
Flags affected	NZC
Comments	ASL A acts on the accumulator.

BCC

Branch if carry clear

Action	Go to specified label or address if C = Ø
Description	If bit C is zero, execution continues at the specified label or address. If bit C is 1 then execution continues at the next instruction.
Addressing modes	Relative
Flags affected	None
Comments	Specified label or address must be in range.

BCS

Branch if carry set

Action	Go to specified label or address if C = 1
Description	If bit C is 1, execution continues at the specified label or address. If bit C is zero then execution continues at the next instruction.
Addressing modes	Relative
Flags affected	None
Comments	Specified label or address must be in range.

BEQ

Branch if equal to zero

Action	Go to specified label or address if Z = 1
Description	If bit Z is 1, execution continues at the specified label or address. If bit Z is zero then execution continues at the next instruction.
Addressing modes	Relative
Flags affected	None
Comments	Specified label or address must be in range.

BIT

Compare memory bits with accumulator

Action	(A) → F (M)
Description	The accumulator is compared with the contents of a memory location. If the two are the same then bit Z is set to 1; if not then bit Z is cleared. Bits 6 and 7 of the data from memory are loaded into bits V and N respectively. The contents of the accumulator remain unchanged.
Addressing modes	Absolute Zero-page
Flags affected	NVZ

BMI

Branch if minus

Action	Go to specified label or address if N = 1
Description	If bit N is 1, execution continues at the specified label or address. If bit N is zero then execution continues at the next instruction.
Addressing modes	Relative

Flags affected	None
Comments	Specified label or address must be in range.

BNE

Branch if not equal to zero

Action	Go to specified label or address if Z = Ø
Description	If bit Z is zero, execution continues at the specified label or address. If bit Z is 1 then execution continues at the next instruction.
Addressing modes	Relative
Flags affected	None
Comments	Specified label or address must be in range.

BPL

Branch if plus

Action	Go to specified label or address if N = Ø
Description	If bit N is zero, execution continues at the specified label or address. If bit N is 1 then execution continues at the next instruction.
Addressing modes	Relative
Flags affected	None
Comments	Specified label or address must be in range.

BRK

Break

Action	STACK ← (PC)+2 STACK ← (F) PCL ← (&FFFE) PCH ← (&FFFF)

Description	This is a software interrupt. The contents of the program counter plus 2 are pushed on the stack, followed by the contents of the flags register. The program counter is then loaded with the contents of locations &FFFE (low byte) and &FFFF (high byte). Bit B is set to 1.
Addressing modes	Implied
Flags affected	B
Comments	Used mainly for error trapping and debugging.

BVC

Branch if overflow clear

Action	Go to specified label or address if V = ∅
Description	If bit V is zero, execution continues at the specified label or address. If bit V is 1 then execution continues at the next instruction.
Addressing modes	Relative
Flags affected	None
Comments	Specified label or address must be in range.

BVS

Branch if overflow set

Action	Go to specified label or address if V = 1
Description	If bit V is 1, execution continues at the specified label or address. If bit V is zero then execution continues at the next instruction.
Addressing modes	Relative
Flags affected	None
Comments	Specified label or address must be in range.

CLC

Clear carry

Action	C ← ∅
Description	Bit C is cleared.
Addressing modes	Implied
Flags affected	C
Comments	Often required before ADC.

CLD

Clear decimal flag

Action	D ← ∅
Description	Bit D is cleared, which means that the processor is in binary mode.
Addressing modes	Implied
Flags affected	D
Comments	Should be used at the beginning of all routines which do not use binary coded decimal.

CLI

Clear interrupt mask

Action	I ← ∅
Description	Bit I is cleared, which enables interrupts.
Addressing modes	Implied
Flags affected	I
Comments	An interrupt is triggered when an external device, such as a printer, requires attention.

CLV

Clear overflow flag

Action	V ← Ø
Description	Bit V is cleared.
Addressing modes	Implied
Flags affected	V

CMP

Compare with accumulator

Action	(A)–data → F
Description	Contents of memory location or immediate data are subtracted from the accumulator. If the result is zero then bit Z is set; if not zero it is cleared. If the result is negative then bit N is set; if positive it is cleared. Bit C is set if the accumulator contents are greater than or equal to the data. The contents of the accumulator remain unchanged; only the flags register is affected.
Addressing modes	Immediate Zero-page Absolute Indexed Indirect/indexed
Flags affected	NZC

CPX

Compare with X register

Action	(X)–data → F
Description	Contents of memory location or immediate data are subtracted from the X register. If the result is zero then bit Z is set; if not zero it is cleared. If the result is negative then bit N is set; if positive it is cleared. Bit C is set if the X register contents are

greater than or equal to the data. The contents of the X register remain unchanged; only the flags register is affected.

Addressing modes
Immediate
Zero-page
Absolute
Indexed
Indirect/indexed

Flags affected NZC

CPY

Compare with Y register

Action (Y)–data → F

Description Contents of memory location or immediate data are subtracted from the Y register. If the result is zero then bit Z is set; if not zero it is cleared. If the result is negative then bit N is set; if positive it is cleared. Bit C is set if the Y register contents are greater than or equal to the data. The contents of the Y register remain unchanged; only the flags register is affected.

Addressing modes
Immediate
Zero-page
Absolute
Indexed
Indirect/indexed

Flags affected NZC

DEC

Decrement memory

Action M ← (M)–1

Description The contents of the specified memory location are decremented by 1.

Addressing modes	Zero-page Absolute Indexed (X only)
Flags affected	NZ

DEX

Decrement X register

Action	X ← (X)–1
Description	The contents of the X register are decremented by 1.
Addressing modes	Implied
Flags affected	NZ
Comments	Enables X to be used as a counter.

DEY

Decrement Y register

Action	Y ← (Y)–1
Description	The contents of the Y register are decremented by 1.
Addressing modes	Implied
Flags affected	NZ
Comments	Enables Y to be used as a counter.

EOR

Logical exclusive-OR

Action	A ← (A) EOR data
Description	Exclusive-OR the contents of memory location or immediate data with the accumulator. Result is placed in the accumulator.

Addressing modes	Immediate Zero-page Absolute Indexed Indirect/indexed
Flags affected	NZ
Comments	See BASIC EOR for truth table.

INC

Increment memory

Action	M ← (M)+1
Description	The contents of the specified memory location are incremented by 1.
Addressing modes	Zero-page Absolute Indexed (X only)
Flags affected	NZ

INX

Increment X register

Action	X ← (X)+1
Description	The contents of the X register are incremented by 1.
Addressing modes	Implied
Flags affected	NZ
Comments	Enables X to be used as a counter.

INY

Increment Y register

Action	Y ← (Y)+1

Description	The contents of the Y register are incremented by 1.
Addressing modes	Implied
Flags affected	NZ
Comments	Enables Y to be used as a counter.

JMP

Unconditional jump

Action	PC ← address
Description	Execution continues at the specified label or address.
Addressing modes	Absolute Indirect
Flags affected	None
Comments	There is no restriction on length of jump; label or address may be anywhere in memory. This is the only instruction which may use straight indirect addressing.

JSR

Jump to subroutine

Action	STACK ← (PC)+2 PC ← address
Description	The contents of the program counter plus 2 are pushed on the stack (this is the address of the instruction following JSR). Execution continues at the specified label or address.
Addressing modes	Absolute
Flags affected	None
Comments	The subroutine to which control is transferred must be terminated by an RTS instruction. JSR is

used whenever you wish to make an operating system call from assembler.

LDA — Load accumulator

Action	A ← data
Description	Load the accumulator with contents of memory location or immediate data.
Addressing modes	Immediate Zero-page Absolute Indexed Indirect/indexed
Flags affected	NZ

LDX — Load X register

Action	X ← data
Description	Load the X register with contents of memory location or immediate data.
Addressing modes	Immediate Zero-page Absolute Indexed (Y only)
Flags affected	NZ

LDY — Load Y register

Action	Y ← data
Description	Load the Y register with the contents of memory location or immediate data.

Addressing modes	Immediate Zero-page Absolute Indexed (X only)
Flags affected	NZ

LSR

Logical shift right

Action	Ø → A or M → C
Description	Shift the contents of the accumulator or memory location right one bit position. Bit Ø falls into the carry (bit C), zero is entered from the left. Result remains in either the accumulator or memory location.
Addressing modes	Zero-page Absolute Indexed (X only) Accumulator
Flags affected	NZC
Comments	LSR A acts on the accumulator.

NOP

No operation

Action	None
Description	Does nothing for two clock cycles.
Comments	Used for timing a program, or to fill in gaps caused by deleted instructions.

ORA

Logical OR

Action	A ← (A) OR data

Description	OR the contents of memory location or immediate data with the accumulator. Result is placed in the accumulator.
Addressing modes	Immediate Zero-page Absolute Indexed Indirect/indexed
Flags affected	NZ
Comments	See BASIC OR for truth table.

PHA

Push accumulator on to stack

Action	STACK ← (A) SP ← (SP)–1
Description	The contents of the accumulator are pushed on to the stack. The stack pointer is decremented. The accumulator contents remain unchanged.
Addressing modes	Implied
Flags affected	None

PHP

Push flags register on to stack

Action	STACK ← (F) SP ← (SP)–1
Description	The contents of the flags register are pushed on to the stack. The stack pointer is decremented. The flags register contents remain unchanged.
Addressing modes	Implied
Flags affected	None

PLA

Pull data from stack into accumulator

Action	A ← (STACK) SP ← (SP)+1
Description	Pull the top byte of the stack into the accumulator. Increment the stack pointer.
Addressing modes	Implied
Flags affected	NZ

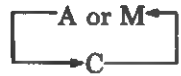
PLP

Pull data from stack into flags register

Action	F ← (STACK) SP ← (SP)+1
Description	Pull the top byte of the stack into the flags register. Increment the stack pointer.
Addressing modes	Implied
Flags affected	NVBDIZC

ROL

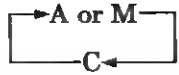
Rotate left

Action	
Description	Rotate the contents of the accumulator or memory location left one bit position. The carry (bit C) is entered from the right, bit 7 falls into the carry.
Addressing modes	Zero-page Absolute Indexed (X only) Accumulator
Flags affected	NZC
Comments	ROL A acts on the accumulator. This is a 9-bit rotation.

ROR

Rotate right

Action



Description — Rotate the contents of the accumulator or memory location right one bit position. The carry (bit C) is entered from the left, bit Ø falls into the carry.

Addressing modes — Zero-page
Absolute
Indexed (X only)
Accumulator

Flags affected — NZC

Comments — ROR A acts on the accumulator. This is a 9-bit rotation.

RTI

Return from interrupt

Action

F ← (STACK)
SP ← (SP)+1
PCL ← (STACK)
SP ← (SP)+1
PCH ← (STACK)
SP ← (SP)+1

Description — Restore the contents of the flags register and the program counter, which were previously stored on the stack. Increment stack pointer.

Addressing modes — Implied

Flags affected — NVBDIZC

Comments — Used to return to the execution of a program, after an interrupt has been dealt with.

RTS

Return from subroutine

Action	PCL ← (STACK) SP ← (SP)+1 PCH ← (STACK) SP ← (SP)+1 PC ← (PC)+1
Description	Restore the contents of the program counter, which were previously stored on the stack, and increment the program counter by 1. Increment the stack pointer.
Addressing modes	Implied
Flags affected	None
Comments	Continues execution from position after subroutine call. Used by the Electron's assembler to return to BASIC.

SBC

Subtract with carry

Action	A ← (A)−data−$\overline{C}$ ($\overline{C}$ is NOT C, which is the borrow.)
Description	Subtract the contents of memory location or immediate data from the accumulator, with borrow. Result is placed in the accumulator.
Addressing modes	Immediate Zero-page Absolute Indexed Indirect/indexed
Flags affected	NVZC
Comments	To subtract without the borrow, bit C must be set beforehand by using SEC.

SEC

Set carry

Action	C ← 1
Description	Bit C is set.
Addressing modes	Implied
Flags affected	C
Comments	Often required before SBC.

SED

Set decimal flag

Action	D ← 1
Description	Bit D is set, which means that the processor is in decimal mode (BCD).
Addressing modes	Implied
Flags affected	D

SEI

Set interrupt mask

Action	I ← 1
Description	Bit I is set, which disables interrupts.
Addressing modes	Implied
Flags affected	I

STA

Store accumulator in memory

Action	M ← (A)
Description	Store contents of the accumulator at the specified memory location. The accumulator contents remain unchanged.

Addressing modes	Immediate Zero-page Absolute Indexed Indirect/indexed
Flags affected	None

STX

Store X register in memory

Action	M ← (X)
Description	Store contents of X register at the specified memory location. The X register contents remain unchanged.
Addressing modes	Zero-page Absolute Indexed (zero-page Y only)
Flags affected	None

STY

Store Y register in memory

Action	M ← (Y)
Description	Store contents of Y register at the specified memory location. The Y register contents remain unchanged.
Addressing modes	Zero-page Absolute Indexed (zero-page X only)
Flags affected	None

TAX

Transfer accumulator to X

Action	X ← (A)
Description	Copy the contents of accumulator into X register. The accumulator contents remain unchanged.
Addressing modes	Implied
Flags affected	NZ

TAY

Transfer accumulator to Y

Action	Y ← (A)
Description	Copy the contents of accumulator into Y register. The accumulator contents remain unchanged.
Addressing modes	Implied
Flags affected	NZ

TSX

Transfer stack pointer to X

Action	X ← (SP)
Description	Copy the contents of the stack pointer into X register. The stack pointer contents remain unchanged.
Addressing modes	Implied
Flags affected	NZ

TXA

Transfer X register to accumulator

Action	A ← (X)

Description	Copy the contents of the X register into the accumulator. The X register contents remain unchanged.
Addressing modes	Implied
Flags affected	NZ

TXS

Transfer X register to stack pointer

Action	S ← (X)
Description	Copy the contents of the X register into the stack pointer. The X register contents remain unchanged.
Addressing modes	Implied
Flags affected	NZ

TYA

Transfer Y register to accumulator

Action	A ← (Y)
Description	Copy the contents of the Y register into the accumulator. The Y register contents remain unchanged.
Addressing modes	Implied
Flags affected	NZ

Appendix A
VDU codes

VDU A is equivalent to **PRINT CHR$ A;**
VDU A,B,C is equivalent to **PRINT CHR$ A; CHR$ B; CHR$ C;**

This chapter is a description of the whole character set from Ø to 255 which can be used with either **VDU** or **PRINT CHR$**. The ASCII table is in Appendix F, and you will see that codes from Ø to 31 give control characters for the screen display; codes 32 to 127 generate visual characters; and the remainder are initially undefined.

Here is each code in detail:

Code	Keyboard	Description
Ø	CTRL @	Does nothing.
1	CTRL A	Reserved.
2	CTRL B	Reserved.
3	CTRL C	Reserved.
4	CTRL D	Allows the text cursor and the graphics cursor to operate independently of one another. (Reverses the action of **VDU 5**.)
5	CTRL E	Causes the text cursor to be joined to the graphics cursor. The text cursor ceases to exist, and characters are printed at the graphics cursor which is positioned using **MOVE**.
6	CTRL F	Allows output to be printed on the screen. (Reverses the action of **VDU 21**.)
7	CTRL G	Causes a short 'beep' from the internal loudspeaker.
8	CTRL H	Moves the text cursor back one space on the screen. Does not delete the previous character.
9	CTRL I	Moves the text cursor forward one space on the screen.

1Ø	CTRL J	Moves the text cursor down one line on the screen. If the cursor is already at the bottom then the screen will scroll up one line.
11	CTRL K	Moves the text cursor up one line on the screen. If the cursor is already at the top then the screen will scroll down one line.
12	CTRL L	Clears the text screen. Same as **CLS**.
13	CTRL M	**VDU13** issued as a command (not in a program), or CTRL M, have exactly the same effect as RETURN. In a program, **VDU13** will move the text cursor to the start of the current line.
14	CTRL N	Puts the display into paged mode. Programs will only be listed to fill the screen, and then the computer will wait until the SHIFT key is pressed before listing another screen full. Used when reading long programs.
15	CTRL O	Cancels paged mode. (Reverses the action of CTRL N.)
16	CTRL P	Clears the graphics screen. Same as **CLG**.
17	CTRL Q	Changes text colour. Same as **COLOUR**.
18	CTRL R	Changes graphics colour, and colour mix. Same as **GCOL**.
19	CTRL S	Assigns any logical colour value to any actual colour. For example, mode 6 normally has two colours only, black and white, assigned the logical colour values Ø and 1. To change Ø (black) to blue, use **VDU19** with the logical colour Ø, and the actual colour 4 (blue). **MODE 6** **VDU19, Ø, 4, Ø, Ø, Ø**

20	CTRL T	Returns all logical colours to normal. (Reverses **VDU19**).
21	CTRL U	CTRL U deletes the whole of the current line being typed. **VDU21**, in a program, disables all output to the screen. This is reversed by **VDU6**.
22	CTRL V	Changes mode. **VDU22,2** is equivalent to **MODE 2**, except that **HIMEM** is not altered.
23	CTRL W	Reprograms a displayed character. 32 user-definable characters are set aside for use with **VDU23**. All the rest of the characters can be redefined if the memory is exploded with ***FX20,1**.
24	CTRL X	Defines a graphics window.
25	CTRL Y	Same as **PLOT**. **VDU25,85,X,Y** is the same as **PLOT 85,X,Y**.
26	CTRL Z	Reverses the effects of **VDU24**, **VDU28**, and **VDU29**. Graphics and text windows both occupy the whole screen; text origin and text cursor are at top left; graphics origin and graphics cursor are at bottom left.
27	CTRL [	Reserved.
28	CTRL ,	Defines a text window.
29	CTRL -	Moves the graphics origin. **VDU29,X,Y** will move **0,0** to position **X,Y**.
30	CTRL .	Homes text cursor to top left of text window.
31	CTRL /	**VDU31,X,Y** is the same as **PRINT TAB(X,Y)**.
32-126	CTRL SPACE to ~	The complete set of ASCII characters.
127	CTRL DELETE	Moves the text cursor back one space on the screen and deletes the character to the current background colour.

128-223	Normally undefined. Can be defined if memory is allocated using ***FX20,1** and **VDU23**.
224-255	User-definable characters. Can be defined using **VDU23**.

Appendix B
Error messages

When the computer is unable to continue executing a program or a command it will tell you by printing a message on the screen. As shown in the section on error trapping, these error messages can be suppressed provided you have written an alternative routine for the computer to follow using **ON ERROR** . . .

As well as an error message, the computer sets two variables each time an error occurs:

ERR gives the error number.

ERL gives the line number at which the error was noticed.

The error messages are listed here in alphabetical order, alongside their error numbers:

Accuracy lost 23

If you try to calculate trigonometric functions with very large angles you will lose a lot of accuracy in reducing the angle to within the range of plus or minus PI radians. When this happens the computer will print the above message, for example:

```
PRINT COS(11111111)
```

Arguments 31

This indicates that there are too many or too few arguments for a given function or procedure.

Array 14

This indicates that the computer expects an array, but cannot find it.

Bad call 3Ø

Incorrect **PROC** or **FN** call.

Bad command 254

Wrongly typed OS command, for example:

***FX20,A**

Bad DIM **10**

Arrays and memory must be dimensioned with a positive number of elements. For example, these will produce errors:

```
DIM array(-10)
DIM P% -2
```

Bad hex **28**

Hex numbers may only consist of 0 to 9 and A to F.

Bad key **251**

Error in ***KEY** command, including running out of space for key string, and attempting to re-define a key while it is in use.

Bad MODE **25**

You cannot change mode inside a **PROC** or **FN**. Nor can you change to a mode which would make **HIMEM** less than **LOMEM**.

Bad Program

There are a number of occasions when the computer checks a program to see where it starts and ends in memory. The above error means that the computer could not follow the program successfully and that it is therefore aborted. This error is untrappable, which means that you cannot find out at which line it occurred, nor can you retrieve any part of the program! It is caused by part of the program becoming over-written either by a mode change or by another program's BASIC variables.

Block? **218**

This is an error generated by the cassette filing system. It means that the computer found a non-consecutive block number. Re-wind the tape a little way and try again.

Byte **2**

Caused in assembly by trying to load a register in immediate mode with a value greater than 255, for example:

```
LDY #266
```

Can't match FOR 33

The control variable associated with **NEXT** is different from that associated with **FOR**.

Channel 222

This error is generated by the cassette filing system when you try to use a file channel number which has not been opened.

Data? 216

This is an error generated by the cassette filing system which means that the computer has missed some data from a block. Re-wind the tape a little and try again.

DIM space 11

There is not enough memory for the array to be dimensioned.

Division by zero 18

You cannot divide by zero.

$range 8

Strings may not be placed in the zero-page of memory using the indirection operator $. This is illegal:

```
$&70 = "error"
```

Eof 223

This error is given by the cassette filing system if the end of file is reached.

Escape 17

The **ESCAPE** key has been pressed.

Exp range 24

You cannot exponentiate a number greater than 88. The following is illegal:

```
A = EXP 9Ø
```

Failed at . . . (line number)

Caused by renumbering a program with a **GOTO** or **GOSUB** to a non-existent line number.

```
15 X = 12
34 GOTO 2ØØ
4Ø END
```

will give the error message:

```
Failed at line 2Ø
```

File? 219

This error is generated by the cassette filing system and means that the computer was given an unexpected file name.

FOR variable 34

The control variable in a **FOR** . . . **NEXT** loop must be numeric, for example:

```
FOR I$ = Ø TO 2Ø
```

is illegal.

Header? 217

This error is generated by the cassette filing system and it means that the computer cannot read the file's header (which contains the name, block number etc). Re-wind the tape a little way and try again.

Index 3

This error occurs during assembly if you use an incorrect index mode, for example:

```
LDA (&7Ø,Y)
```

LINE space

The computer has run out of memory for you to type any extra lines into a program.

Log range **22**

You cannot find the log of a negative or zero number.

Missing , **5**

This means that the computer expected to find a comma in the line, but didn't do so, for example:

```
C$ = LEFT$(Z$)
```

Missing " **9**

This means the computer expected to find a quote, for example:

```
CHAIN "MARSLANDER
```

Missing) **27**

This means the computer expected to find a closing bracket, for example:

```
PRINT TAB(6,16
```

Missing # **45**

This means the computer expected to find a hash, for example:

```
?I% = BGET file
```

Mistake **4**

This means that the computer could not understand the instruction, for example:

```
1Ø PRIT
```

–ve root **21**

You cannot find the square root of a negative number. This may also occur with **ASN** and **ACS**.

No GOSUB **38**

The computer encounters **RETURN** without having passed a **GOSUB**.

No FN **7**

The computer encounters the end of a function definition without having passed the **DEF FN**, for example:

```
= X
```

No FOR **32**

The computer encounters **NEXT** without having passed the **FOR**.

No PROC **13**

The computer encounters **ENDPROC** without having passed the **DEF PROC**.

No REPEAT **43**

The computer encounters **UNTIL** without having passed the **REPEAT**.

No room **Ø**

When a program is running, the computer uses the area of memory between **LOMEM** and **HIMEM** to store the BASIC variables. If there is insufficient room to store any more of these variables then the above error is given. This most commonly occurs with programs which use arrays in conjunction with a large screen mode (Ø, 1, 2, or 3).

No such FN/PROC **29**

The computer encounters an **FN** or a **PROC** for which it can find no definition.

No such line **41**

Electron BASIC does not allaow you to **GOTO** or **GOSUB** a line number which does not exist.

No such variable **26**

All variables must be declared, either globally by assigning them a value or locally by using **LOCAL**. If the computer encounters an un-declared variable it gives the above error. This error is also given in assembler when the computer encounters a forward reference to a label.

No TO **36**

TO is omitted from the **FOR . . . NEXT** loop:

```
FOR I=0
```

Not LOCAL **12**

Local variables may be declared only within an **FN** or a **PROC**.

ON range **40**

The control variable for **ON GOTO** or **ON GOSUB** is either less than 1 or is greater than the number of entries in the list of line numbers. For example, the following will not work if **destination = 3**:

```
ON destination GOTO 60,210
```

because there are only two destinations. This error may be accounted for by using **ELSE**:

```
ON destination GOSUB 70,90 ELSE . . .
```

ON syntax **39**

The word **ON** must be followed either by **ERROR**, or by a numeric variable and **GOTO** or **GOSUB**. The following will give an error:

```
ON direction PRINT
```

Out of DATA **42**

The computer encounters a **READ** instruction for which it cannot find an entry in the **DATA** list. **RESTORE** can be used to move the data-pointer back to the start of a **DATA** list.

Out of range **1**

Branch instructions in assembler can access not farther than 127 bytes forwards or 128 bytes backwards. To branch outside these limits you must use **JMP** or **JSR**.

Silly **Ø**

Given by the automatic line numbering system **AUTO** or the line renumbering system **RENUMBER** if you attempt to use a step size of less than 1 or more than 255.

String too long **19**

The maximum length of a string is 255 characters.

Subscript **15**

An array subscript is out of range, either less than Ø or greater than the value declared in **DIM**.

Syntax **22Ø**

Bad syntax in the cassette filing system.

Syntax error **16**

A statement is incorrectly terminated, for example:

```
LIST. 5Ø
```

Too big **2Ø**

The computer calculates a number which is too big or too small to be represented.

Too many FORs **35**

FOR . . . NEXT loops may be nested to a depth of 1Ø, and the control variables must all be different.

Too many GOSUBs **37**

GOSUB . . . RETURN loops may be nested to a depth of 26.

Too many REPEATs **44**

REPEAT . . . UNTIL loops may be nested to a depth of 2Ø.

Type mismatch **6**

You cannot assign a string to a numeric variable or a number to a string variable.

Appendix C
Operating system calls

The computer's operating system is a program, stored in read-only memory, which runs continually, sorting out what goes where and when. Some parts of the operating system can be accessed from BASIC, and these instructions all begin with an asterisk. When executing BASIC, the computer will pass these instructions straight to the operating system.

Here is a list of available commands:

CAT** Catalogues all file names on cassette, and displays them on the screen. Can be shortened to **.

***SAVE** Saves a section of memory onto tape. ***SAVE "File" 1000 1FFF 102A** will save a page of memory, called File, from address **&1000** to **&1FFF**, and an execution address (for use by ***RUN**) of **102A**. If the execution address is omitted, it is assumed to be equal to the start address.

***RUN** Loads and runs a program stored by ***SAVE**. ***RUN "File"** will load and run the example given in ***SAVE** above.

***LOAD** Loads a file and stores it in memory at a specified address. ***LOAD "Game" 2000** will load from tape a file called Game, and store this at location **&2000**.

***OPT** Determines the computer's reaction to errors during cassette operations.

***OPT1,X** Controls the error messages given.

X = 0 Gives no messages.
X = 1 Gives short messages (as normal).
X = 2 Gives long messages, including load and execution addresses.

***OPT2,X** Controls the computer's action.

X = 0 Lets the computer ignore all errors, and carry on regardless. Messages can still be given.
X = 1 The computer asks you to try again by re-winding the tape (as normal).
X = 2 The computer aborts the operation.

***OPT3,X** Sets the inter-block gap – the time delay between each page of memory stored on the tape. X determines the gap in tenths of a second. This only applies to **PRINT#** and **BPUT#**. The gap on **SAVE** is fixed at Ø.6 seconds.

***OPT** On its own sets all the values to normal.

***SPOOL** Used for saving a program listing or results (as long as they are ASCII characters) to cassette or disc as a text file. Thus ***SPOOL FRED** opens a file called FRED on either tape or disc. Whatever characters appear on the screen after that (eg a program listing called up by **LIST**, or one you enter on the keyboard) will be saved to FRED as a text file. To close the file, type ***SPOOL** **RETURN** at the end.

***EXEC** Loads a file from tape, as input rather than as a program. Used for loading a file which has been ***SPOOL**ed.

***MOTOR** Used to turn the cassette motor relay on or off. ***MOTOR0** for off, ***MOTOR1** for on.

***KEY** Programs a user-defined function key.

***FX** A family of operating system commands which are described in Appendix D.

***HELP** Gives version numbers of current software.

Appendix D
*FX calls

*FX calls provide a variety of controls for operating system functions such as auto-repeat, flash-rate, buffer-flushing, memory allocation etc, etc. The following is a description of all *FX calls available from BASIC.

Call	Description
*FX0	Prints a message on the screen telling which operating system you have. Operating systems are updated from time to time by the manufacturer.
*FX4	Controls the operation of the four 'arrow' keys and COPY.

*FX4,1 disables their editing function, and causes them to generate ASCII codes, just like any other key:

COPY	135
'left-arrow' key	136
'right-arrow' key	137
'down-arrow' key	138
'up-arrow' key	139

*FX4,2 allows the five keys to be user-programmable. Their key values become:

COPY	*KEY11
'left-arrow' key	*KEY12
'right-arrow' key	*KEY13
'down-arrow' key	*KEY14
'up-arrow' key	*KEY15

*FX4,0 resets the keys to their normal function of editing. It reverses *FX4,1 and *FX4,2.

*FX9 *FX10	Used to set the flash-rate of flashing colours. *FX9 controls the duration of the first colour, *FX10 the duration of the second.

```
*FX9,25
*FX10,25
```

Will set each colour to stay on for equal time of 25 fiftieths (½) of a second. These are in fact the normal values when the machine is switched on.

You could change them to:

```
*FX 9,40
*FX10,20
```

which will make the first colour stay on for twice as long as the second; for 4/5 and 2/5 of a second respectively. If one duration is set to Ø, the other colour will stay on all the time.

***FX11** Sets the delay, when a key is pressed, before the auto-repeat comes into action.

***FX11,50** will set the delay to 5Ø hundredths (½) of a second.

***FX11,0** turns off the auto-repeat altogether.

***FX12** Sets the period of auto-repeat.

***FX12,10** sets the auto-repeat to 1Ø hundredths (1/1Ø) of a second between characters, giving 1Ø characters per second.

***FX12,0** resets both ***FX11** and ***FX12** to their normal values. As an example, type in the following:

```
*FX12,1     RETURN
*FX11,1     RETURN
```

and now try typing in anything at all!

***FX13** Disable/enable events.

***FX14** See chapter on Assembly Language.

***FX15** Flushes (empties) certain buffers (short term memories).

***FX15,0** flushes all buffers.
***FX15,1** flushes the currently selected input buffer.

***FX18** Resets all the user-programmable keys to empty.

***FX19** Makes the computer wait for the start of the next screen display frame.

***FX20** Determines the amount of memory which can be set aside for character definitions.

***FX20,0** implodes the character definitions. This means that the extra memory set aside for extra character definitions by ***FX20,1** to ***FX20,6** is returned to the user for storing BASIC programs.

***FX20,1** to ***FX20,6** explodes the memory to store groups of 32 extra character definitions. All characters with ASCII codes 32 to 255 may be user defined. As described in chapter 21, ASCII codes 128 to 255 may be defined without using a ***FX20** command, but only 32 consecutive characters can be defined. ***FX20,1** to ***FX20,6** take chunks of memory from the BASIC program storage area to hold specific definitions, and this is shown in the table below.

ASCII code	*FX	Memory allocation
128–159	*FX20,0	&C00 to &CFF
160–191	*FX20,1	OSHWM to OSHWM + &FF
192–223	*FX20,2	OSHWM + &100 to OSHWM + &1FF
224–255	*FX20,3	OSHWM + &200 to OSHWM + &2FF
32– 63	*FX20,4	OSHWM + &300 to OSHWM + &3FF
64– 95	*FX20,5	OSHWM + &400 to OSHWM + &4FF
96–127	*FX20,6	OSHWM + &500 to OSHWM + &5FF

OSHWM stands for Operating System High Water Mark, and means the point where the memory (from &0000) occupied by the operating system ends, and the memory occupied by BASIC programs begins. Turn to chapter 23 for the computer's memory map. The OSHWM normally sits at &E00, but this will change when a software expansion has been fitted, eg a disc filing system.

If you explode the memory allocation in this way you must remember to reset **PAGE** higher up the memory. A program stored at &E00 may be lost.

***FX21** Flushes (empties) certain buffers (short term memories).

***FX21,0** flushes the keyboard buffer.

***FX21,4** flushes sound channel 0.

***FX21,5** flushes sound channel 1.

***FX21,6** flushes sound channel 2.

***FX21,7** flushes sound channel 3.

***FX124** Used to reset the flag at memory location &00FF which tells when an ESCAPE has occurred.

***FX125** Sets the above-mentioned flag. Has similar effect to pressing the **ESCAPE** key.

FX126** Used when reading characters from an input stream using OSRDCH.FX126** acknowledges the detection of an ESCAPE.

***FX138** Used to insert a character into the keyboard buffer.

***FX138,0,X** will insert **CHR$ X**.

***FX225** Disables all the user-definable keys.

***FX226** Disables **FUNC** A to **FUNC** P.

***FX227** Disables **FUNC** Q onwards.

With a parameter value other than the two given above, ***FX225-227** will cause **FUNC** keys to give ASCII codes. For example, ***FX226,224** will cause **FUNC** A to give 224, **FUNC** B to give 225, etc. At this setting (***FX226, 224**), **FUNC** A onwards will give the standard range of user-defined characters direct from the keyboard. Any number from 2 to 255 may be used as the parameter for ***FX226**, and determines the base code, which will be given by **FUNC** A.

Appendix E
Fast and efficient programs

Programming

The book which accompanies this one, called *Start Programming with the Electron,* is intended as a guide to writing programs in Electron BASIC. Once you start writing your own programs, however small they may be, please make good use of the *Start Programming with the Electron* book. It will help you to develop a good style and help you to avoid getting into bad habits. You will find the programs referred to in the book on the Introductory Cassette.

Speeding up programs

There may be times where the program must run as quickly as possible, and here are a few tips for increasing execution speed.

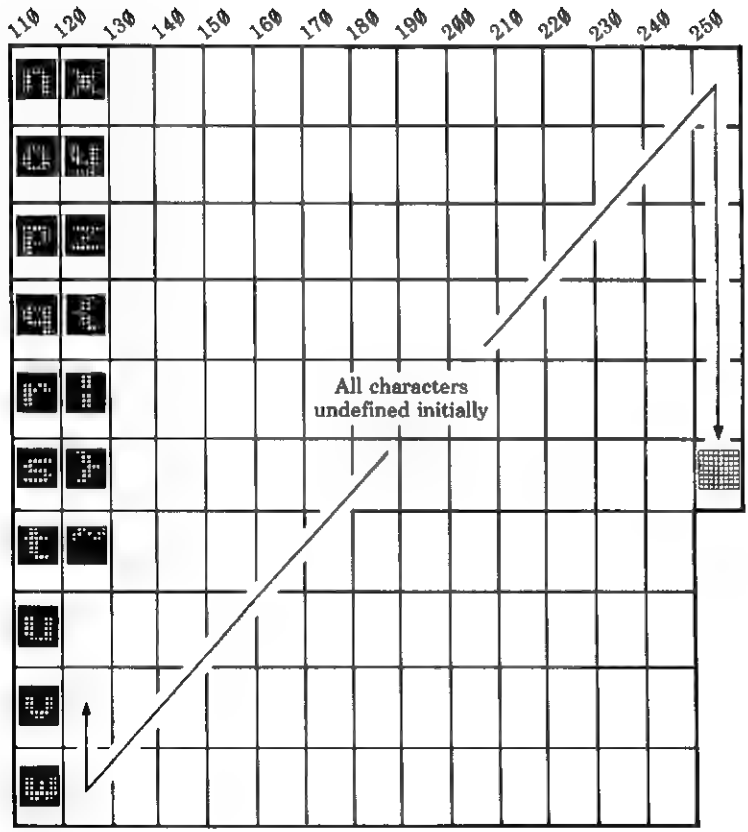
- Use integer variables rather than real variables wherever possible.
- Use integer division (**DIV**) rather than normal division (/).
- Use integer arrays rather than real arrays.
- Start your variable names with different letters of the alphabet.
- Omit the control variable after the instruction **NEXT**.
- **REPEAT . . . UNTIL** loops are faster than **IF . . . THEN GOTO** loops.
- Use procedures instead of **GOSUB**s.
- Use as few line numbers as possible by using a colon to separate each statement.
- Leave out as many spaces as possible, without confusing the computer.
- Omit **REM** statements.

Some of these tips will have the effect of making your programs less readable. This is not a good thing, but in cases where speed is all important, you may find that you will have to reach a compromise between readability and execution speed.

Appendix F
ASCII displayed character set and control codes

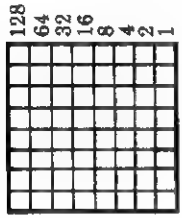
	Ø	1Ø	2Ø	3Ø	4Ø	5Ø	6Ø	7Ø	8Ø	9Ø	1ØØ
Ø	Nothing	Down	Default logical colours	Move text cursor to ØØ							
1	Next to printer	Up	Disable VDU	Move text							
2	Start printer	Clear text	Select mode								
3	Stop printer	Start of line	Reprogram characters								
4	Separate cursors	Paged mode	Nothing								
5	Join cursors	Scroll mode	Plot								
6	Enable VDU	Clear graphics	Default text/ graphics areas								
7	Beep	Define text colour	Define graphics area								
8	Back	Define graphics colour	Define text area								
9	Forward	Define logical colours	Define graphics origin								

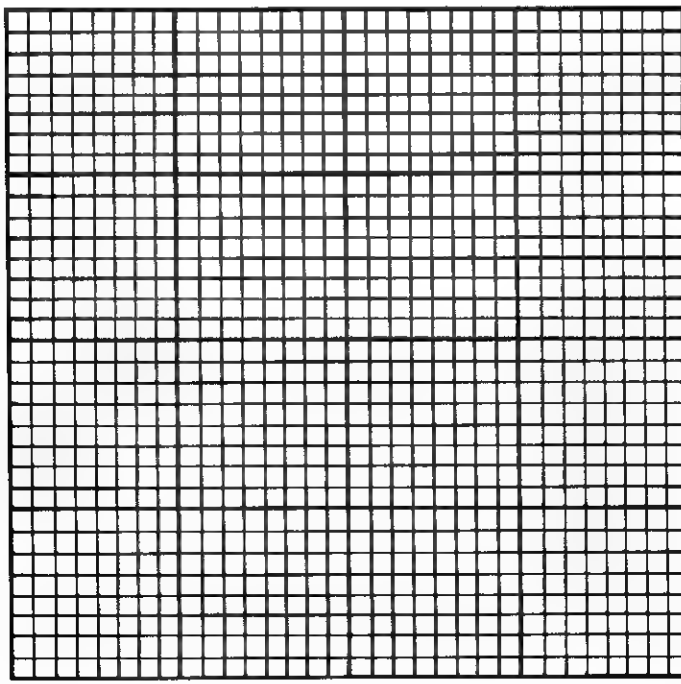
Each displayed character consists of 8 rows of 8 dots


11Ø
12Ø
13Ø
14Ø
15Ø
16Ø
17Ø
18Ø
19Ø
2ØØ
21Ø
22Ø
23Ø
24Ø
25Ø
All characters
undefined initially

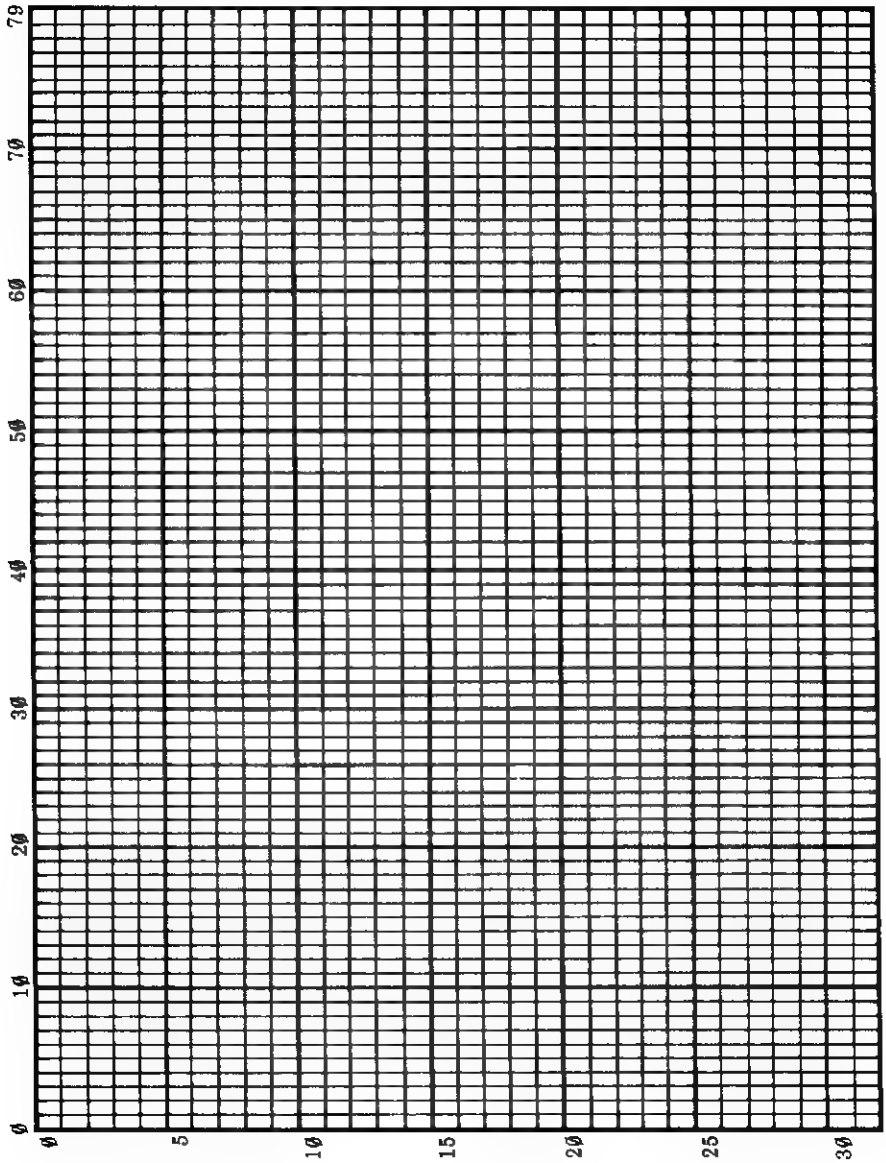
Appendix G
Text and graphics planning sheets

User defined planning sheet

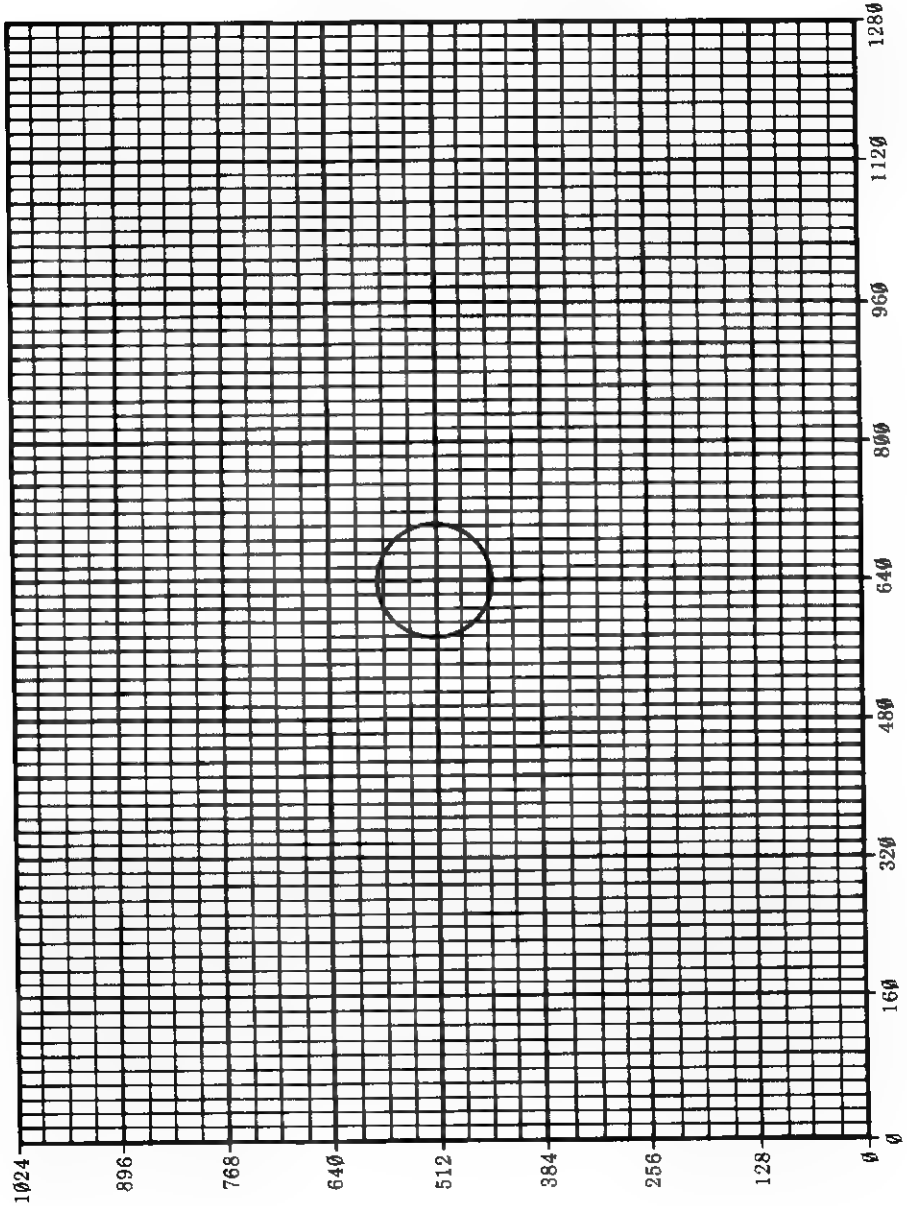




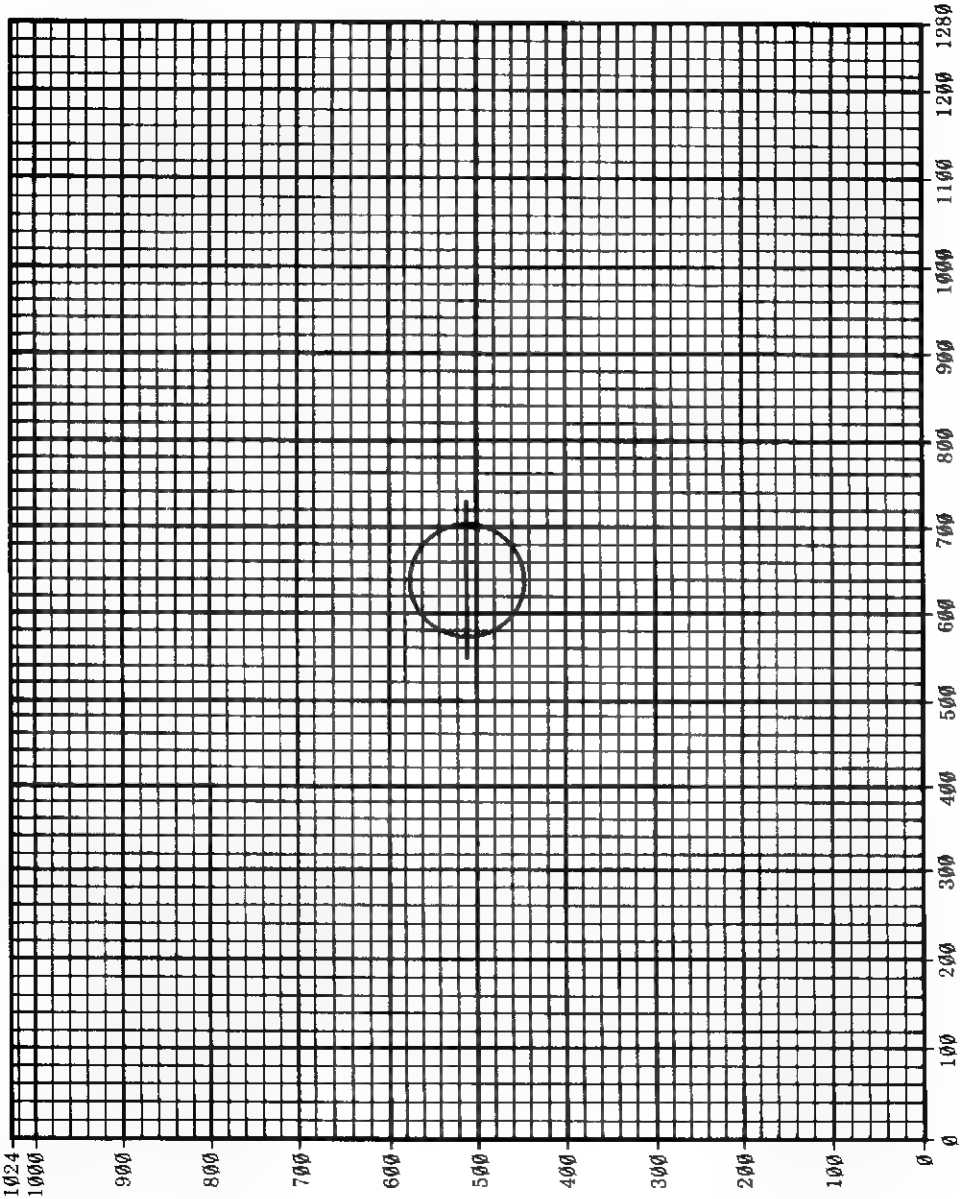
Text planning sheet



Graphics planning sheet 1 (grid related to character positions)



Graphics planning sheet 2 (decimal)



Electron User Guide Errata

p13 Key Z rotates the craft anticlockwise, not key Y.

p59 Listing at bottom of page: line 20 should read
20 PRINT A$;A$;A$

p60 Line 30 of the listing should read
30 PRINT X,X,X'A4,A$,A$

p62 2(ii): @% = &0090A

p66 Last sentence should read – If you don't press a key in time, I N K E Y **S** will give a null string (ASCII value —1).

p68 First sentence should read – Programs and parts of programs can be made to execute over and over again, either continuously, a specified number of times or until a certain event happens.

p75 Third line from top should read
40 UNTIL FALSE
Second sentence of text should read – You will see that only real numbers
-1<X<1 are treated as FALSE, all other values being TRUE.
Line 40 of the second listing (12 lines down) should read
40 UNTIL 0
Beginning of last paragraph on page should read —This will be false, because...

p84 Line 10 of the listing should read
10 PRINT 'SCREENFUL';

p86 ELSE may be used with ON. .GOTO and ON.. .GOSUB to trap any value which is out of range. For example
ON N GOSUB 100,200,70,260 ELSE 999
will GO SUB 999 for any value of N outside the range ito 4.

p98 First line of second paragraph should read – Each pixel has a logical colour... Fifth sentence in this paragraph should read – EOR means 'exclusive OR', which is a non-equivalence comparison giving a 1 result when the compared bits are different (1 and 0, or 0 and 1), and a 0 result when the compared bits are the same (0 and 0, or land 1).

p100 Ignore references to *FX 13.

p101 Four lines from top of page should read
VDU24,a; b; c; d;
where a , b and c , d are the coordinates of the .

p129 2 lines from top of page should read
*SAVE filename" SSSS FFFF EEEE RRRR **RETURN**

Insert the following underneath the above —RRRR is the hex address at which the program will reload.

p131 KEY 3 definition should read
*KEY3 '10 REPEAT |M 20 PRINT CHR$(RND(95)+31); |M 30 UNTIL VPOS=24 |M RUN |M

p173 OPEN U P : insert following note—Note that for the cassette filing system, this command is equivalent to OPEN I N (ie input only).

p195 See above note.

p207 Middle of page, end of paragraph should read – contents of memory location &BBAA.

p285 Swap text in boxes 24 and 27.

p286 Character number 127 should have the following description – backspace and delete.