

ZX Spectrum +2

sinclair



Introduction

Sinclair ZX Spectrum +2 128K Home Computer

A culmination

Built on the outstanding success of the established ZX range of computers - the original Spectrum, the Spectrum + and the Spectrum 128, we now proudly present the ZX Spectrum +2, a machine that combines the ingenuity of Sinclair technology with Amstrad's expertise in integration and engineering reliability.

Software compatibility

The +2 may be used with software written for the earlier models in the ZX Spectrum range. This means that a vast quantity of software already exists for the +2. There are literally thousands of titles available covering every conceivable application: games, utilities, music, scientific, educational and many many more.



When choosing software, always look out for the 'Sinclair Quality Control' logo on the software package itself.

We recommend that you buy software only from manufacturers operating under this scheme which was set up as a control against incompatible or misleadingly-labelled software.

About this book

This book is not intended to be an exhaustive guide to every aspect of computing on the +2. If you need to delve deeper, then there are many existing publications for the Spectrum + and Spectrum 128 computers which will serve this purpose admirably and provide you with all you need to know about ZX Spectrum computers in general, and about Sinclair BASIC.

If all you wish to do, however, is find out how to set up the computer, connect add-ons, learn the fundamentals of BASIC programming, and load software and games, then this book will prove entirely adequate for your requirements.

AMSTRAD

CONSUMER ELECTRONICS PLC.

© Copyright 1986 - AMSTRAD Consumer Electronics plc.

Neither the whole nor any part of the information contained herein, nor the product described in this manual, may be adapted or reproduced in any material form except with the prior written approval of AMSTRAD Consumer Electronics plc. ('Amstrad').

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 Amstrad in good faith.

All maintenance and service on the product must be carried out by Sinclair authorised dealers. Amstrad cannot accept any liability whatsoever for any loss or damage caused by service or maintenance by unauthorised personnel. This guide is intended only to assist the reader in the use of the product, and therefore, Amstrad 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 guide or any incorrect use of the product.

We ask that all users take care to submit their user registration/guarantee cards.

All correspondence relating to the product or to this manual should be addressed to:

Sinclair Computers Division
Amstrad Consumer Electronics plc
Brentwood House
169 Kings Road
BRENTWOOD
Essex CM14 4EF

First Published 1986
Second Edition 1986

Written by Ivor Spital and Rupert Goodwins

Extracts from the book 'ZX Spectrum BASIC programming' written by Steven Vickers and Robin Bradbeer

Typeset and published by Amstrad

Amstrad is a registered trademark of Amstrad Consumer Electronics plc. Unauthorised use of the trademark or word Amstrad is strictly forbidden.

IMPORTANT

You must read this...

1. Always connect the mains lead of power supply unit to a 3-pin plug following the instructions given in chapter 1.
2. Do not attempt to connect the power supply unit to any mains supply other than 220-240V AC 50Hz.
3. Whenever you have finished using the **+2**, ALWAYS disconnect the power supply unit from the mains supply socket.
4. There are no user serviceable parts inside the equipment - DO NOT ATTEMPT TO GAIN ACCESS INSIDE THE POWER SUPPLY UNIT - THERE ARE HIGH VOLTAGES INSIDE. Refer all servicing to qualified service personnel.
5. Do not block or cover the ventilation slots in the equipment.
6. Do not use or store the equipment in excessively hot, cold, damp, or dusty areas.
7. Never plug in (or unplug) any device from the **EXPANSION I/O** socket while the **+2** is switched on - doing so will probably damage both the **+2** and the expansion device.
8. After you have switched off your TV (or VDU monitor), do not immediately disconnect the **+2** - wait a few seconds or so.
9. Do not switch off the **+2** (or switch on or off any peripheral devices connected to the **+2**) while there is a program or data in the memory that you wish to keep - doing so may make the **+2** 'crash', losing the program or data.

Contents

Chapter 1	7
Open the box	
Unpacking	
Fitting a mains plug	
Setting up	
Chapter 2	11
Operating your +2	
Switching on	
Tuning-in your TV	
Using the +2	
The opening menu	
Chapter 3	17
How to load Spectrum 128 software	
Loading software	
Abandoning loading	
Resetting the +2	
Chapter 4	19
How to load Spectrum 48 software	
Loading software	
Abandoning loading	
Resetting the +2	
Chapter 5	21
Introduction to BASIC	
Chapter 6	23
Using 128 BASIC	
The editor	
The edit menu	
Renumbering a BASIC program	
Swapping screens	
Listing to the printer	
Typing in a program	
Moving the cursor	
Running a program	
Commands and instructions	

Chapter 7 29
Using 48 BASIC

Using the #2 as a 48K Spectrum
Entering 48 BASIC mode
The keyboard under 48 BASIC
Program entry
Editing the current line

Chapter 8 37
A complete guide to BASIC programming

Part 1 - Introduction
Part 2 - Simple programming concepts
Part 3 - Decisions
Part 4 - Looping
Part 5 - Subroutines
Part 6 - Data in programs
Part 7 - Expressions
Part 8 - Strings
Part 9 - Functions
Part 10 - Mathematical functions
Part 11 - Random numbers
Part 12 - Arrays
Part 13 - Conditions
Part 14 - The character set
Part 15 - More about PRINT and INPUT
Part 16 - Colours
Part 17 - Graphics
Part 18 - Motion
Part 19 - Sound
Part 20 - Datacorder operations
Part 21 - Printer operations
Part 22 - Other peripherals
Part 23 - IN and OUT
Part 24 - The memory
Part 25 - The system variables
Part 26 - Using machine code
Part 27 - Spectrum character set
Part 28 - Reports
Part 29 - Reference information
Part 30 - The BASIC
Part 31 - Example programs
Part 32 - Binary and hexadecimal

Chapter 9	193
Using the calculator	
Selecting the calculator	
Entering numbers	
Running total	
Using built-in mathematical functions	
Editing the screen	
Assigning variables	
Exit-ing from the calculator	
Chapter 10	197
Connecting peripherals to your +2	
Joystick(s)	
VDU Monitor	
Amplifier	
Printer	
Serial devices	
MIDI device	
Keypad	
Interface One and microdrives	
Other expansion devices	
Index	203

Chapter 1

Open the box

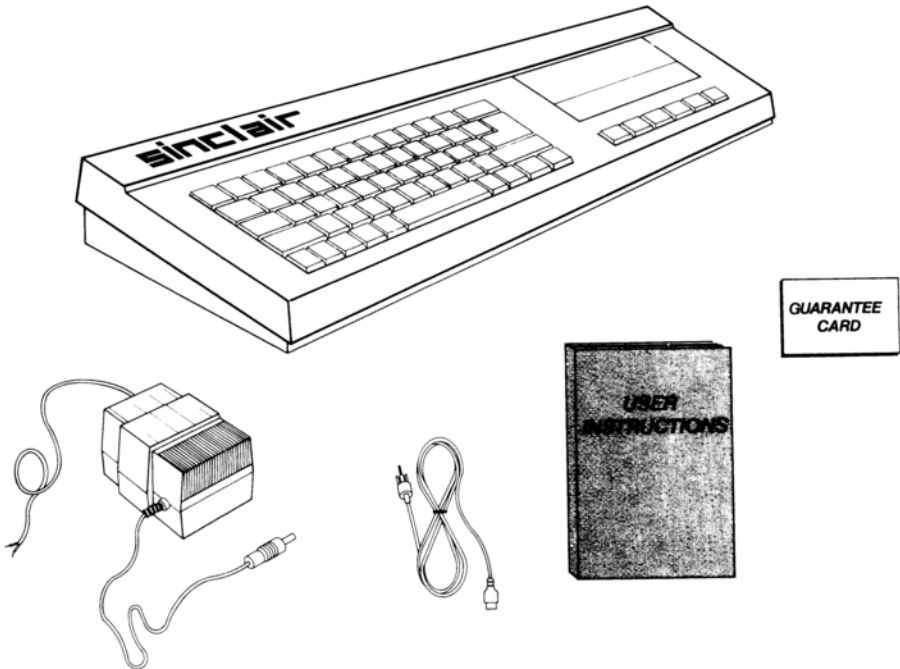
Subjects covered...

Unpacking
Fitting a mains plug
Setting up

Unpacking

Inside the carton, you'll find the following...

The Spectrum +2 computer
The power supply unit
The aerial lead
This manual (together with your user registration/guarantee card)



Fitting a mains plug

The power supply unit for the Spectrum **+2** operates from a 220-240 Volt AC 50Hz mains supply.

Fit a proper mains plug to the mains lead of the power supply unit. If a 13 Amp (BS1363) plug is used, a 3 Amp fuse must be fitted. The 13 Amp fuse supplied in a new plug must NOT be used. If any other type of plug is used, a 5 Amp fuse must be fitted either in the plug or adaptor or at the distribution board.

IMPORTANT - The wires in this mains lead are coloured in accordance with the following code:

Blue: Neutral
Brown: Live

As the colours of the wires in the mains lead of this apparatus may not correspond with the coloured markings identifying the terminals in your plug, proceed as follows...

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

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

Disconnect the mains plug from the supply socket when not in use.

Do not attempt to remove any screws, nor open the casing of the power supply unit. Always obey the warning on the rating label of the power supply unit...

WARNING: LIVE PARTS INSIDE - DO NOT REMOVE ANY SCREWS

Setting up

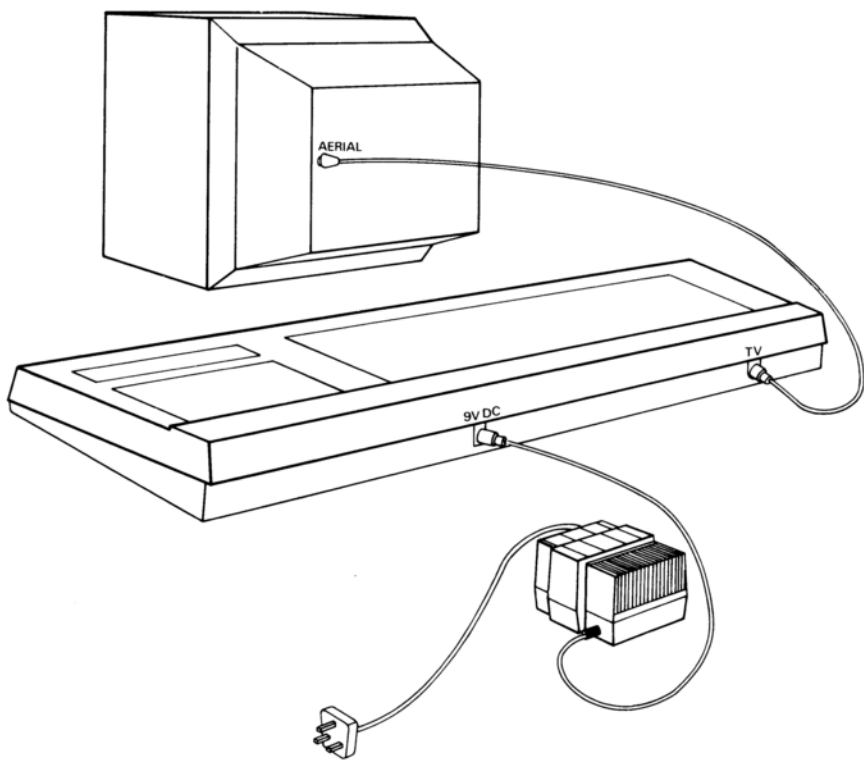
We will now set up the standard **+2** system. All you need (other than the items you unpacked) is a standard TV set (UHF). You can use a colour or black-and-white TV, but of course, with the latter you will not be able to enjoy the full colour capabilities of your **+2**.

Note that if you wish to attach add-ons, or *peripherals*, (such as joystick(s), microdrive(s), a monitor, keypad, audio amplifier, MIDI device, printer or other serial/expansion devices) to your **+2** system, you should turn to chapter 10 (Connecting peripherals to your **+2**).

Place the **+2** computer on a suitable flat surface, ready to be connected to your TV. Next, remove any plug which is already connected to the aerial socket at the back of the TV. Using the aerial lead provided with your **+2**, insert the *larger* plug into the TV's aerial socket, and insert the *smaller* plug into the socket marked **TV** at the back of the **+2**.

Finally, insert the small plug coming from the power supply unit into the socket marked **9V DC** at the back of the **+2**.

The **+2** system is now ready to be switched on.



The standard +2 system setup

Chapter 2

Operating your +2

Subjects covered...

- Switching on
- Tuning-in your TV
- Using the +2
- The opening menu

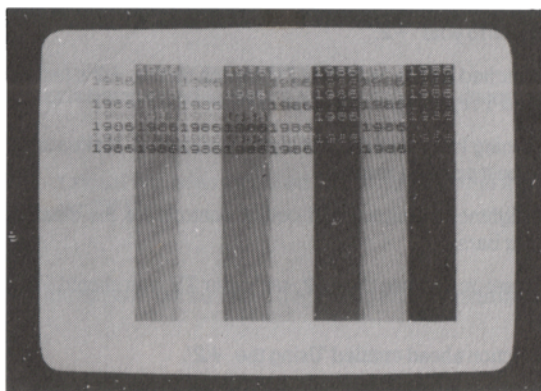
Switching on

Connect the mains plug of the power supply unit to the mains supply socket, and switch on the socket-switch (if necessary). The **ON** indicator lamp (on the top panel of the +2) should illuminate.

Now switch on your TV. On the screen you will probably see either a faint TV picture or just random 'white noise' and hear a loud 'hissing' sound from the TV's speaker. Adjust the TV's volume control until the sound is at a comfortable listening level. The next thing to do is set up the +2 ready for tuning-in.

Preparing to tune-in your TV

The +2 is capable of generating its own test signal, enabling you to tune-in the TV accurately. The test signal consists of sixteen vertical colour bars (containing text characters) which appear on the TV screen, and a repeating tone which is reproduced through the TV's speaker. (If you are using a black-and-white TV, then the colour bars appear as varying shades of grey). You will see and hear the test signal when you have completed the tuning-in of your TV (described ahead).



Switch on the test signal by holding down the **[BREAK]** key (at the top right of the keyboard) and while it is held down, press and release the **RESET** button (at the left hand side of the **+2**). Keep the **[BREAK]** key held down for a few seconds longer, then release it. The test signal will now be generated by the **+2**, and you should proceed to tune-in your TV, as now described.

Push-button TV channel selectors

If your TV *doesn't* have push-button channel selectors, then skip to the section ahead entitled 'Manual tuning'.

If your TV *does* have push-button channel selectors, press one of them to select a *spare* channel (ie. one not normally used for receiving TV or video programmes). Note that if your TV is equipped with an *AFC* (or *AFT*) switch, then this should be set to the *off* position.

Using the tuning control that corresponds to the selected channel, tune-in to the test signal (shown on the previous page). Make sure that both picture *and* sound are tuned-in for the best possible results.

When you are satisfied with the tuning, then you may (if your TV is so equipped) set the *AFC* (or *AFT*) switch to the *on* position.

Finally, adjust the TV's brightness, contrast and colour controls for the clearest display of the text characters within the colour bars.

Now that you have tuned-in one of the TV's push-button channel selectors *specifically* for the **+2**, you may thereafter select that particular channel whenever you wish to use the **+2** with your TV.

You may now skip to the section ahead entitled 'Using the **+2**'.

Manual tuning

If your TV isn't equipped with push-button channel selectors, then you will have to use the TV's manual tuning knob to tune-in to your **+2**.

Having connected and switched on the **+2** and TV, switch on the **+2**'s test signal as described in the previous section entitled 'Preparing to tune-in your TV'.

Tune-in the TV's manual tuning knob until the test signal is received. Make sure that both picture *and* sound are tuned-in for the best possible results.

Finally, adjust the TV's brightness, contrast and colour controls for the clearest display of the text characters within the colour bars.

Each time that you wish to set up and use the **+2** with your TV, you should follow the above manual tuning procedure.

You may now skip to the section ahead entitled 'Using the **+2**'.

Having problems?

If you have tuned-in your TV satisfactorily, you may now skip to the section ahead entitled 'Using the +2'.

If you are unable to tune-in your TV, the following check list may help you to ascertain where the problem lies, and what remedial action you can take.

1. Problem...

The **ON** indicator lamp (on the top panel of the +2) is not illuminated.

Action...

- * Check power supply unit is plugged into computer.
- * Check mains plug of power supply unit is plugged into mains supply socket.
- * (If mains supply socket is switched) - Check supply socket switch is on.
- * Check connections and fuse in mains plug.

2. Problem...

The **ON** indicator lamp is illuminated, but no signal whatsoever can be tuned-in on the TV.

Action...

- * Check TV is set up and working correctly.
- * Check TV is standard UHF type (colour or black-and-white).
- * Check aerial lead (supplied) is connected from computer to TV aerial socket.
- * (If you have push-button channel selectors) - Check you are tuning-in the channel you selected.

3. Problem...

Only a poor signal from the computer can be tuned-in on the TV.

Action...

- * Check TV is set up and working correctly.
- * Check aerial lead (supplied) is fully plugged into computer and TV aerial socket.
- * (If TV is so equipped) - Check AFC (or AFT) switch is set to off position.
- * Check tuning-in has been carried out as accurately as possible.

4. Problem...

Signal from the computer is being tuned-in, but it's not the test signal described above.

Action...

- * Check computer's test signal has been switched on (as described in the previous section entitled 'Preparing to tune-in your TV').

5. Problem...

The test signal colour bars appear, but no sound (repeating tone) is audible from the TV's speaker.

Action...

- * Check TV's volume control is not at minimum.
- * Check tuning-in has been carried out as accurately as possible.

6. Problem...

The test signal sound (repeating tone) can be heard, but no colour bars can be seen on the TV.

Action...

- * Check TV's brightness, contrast and colour controls are not at minimum.
- * Check tuning-in has been carried out as accurately as possible.

7. Problem...

The test signal colour bars and sound are tuned-in, but none of the text characters can be read.

Action...

- * Check tuning-in has been carried out as accurately as possible.
- * Check TV's brightness, contrast and colour controls are adjusted for best results.

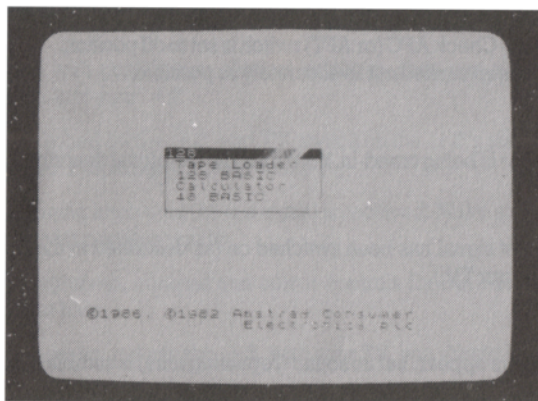
If you cannot identify the cause of your problem, try carrying out the entire procedure (from the beginning of this chapter) again. If the problem still persists, contact your Sinclair dealer.

Using the +2

The **+2** system should now be fully set up, with the test signal colour bars on the screen, and the repeating tone coming from the TV's speaker.

We will now switch off the test signal and start using the **+2**. Press and release the **RESET** button (at the left hand side of the **+2**). The test signal will disappear from the screen, and in its place will be the 'opening menu'.

The opening menu



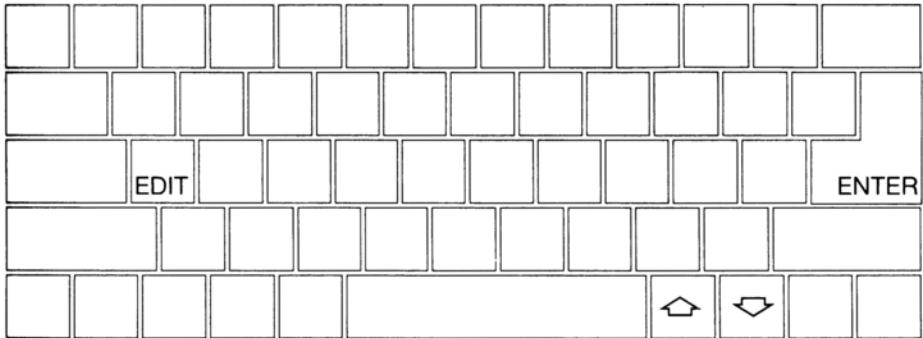
The opening menu appears whenever you first plug in and switch on the **+2**, or whenever you press and release the **RESET** button.

As its name suggests, the opening menu offers you a selection of options. You can choose from one of the four options which appear within the central box on the screen. These are...

- Tape Loader - Choose this option if you wish to load Spectrum 128 software.
- 128 BASIC - Choose this option if you wish to use the +2 for BASIC programming.
- Calculator - Choose this option if you wish to use the +2 as a calculator only.
- 48 BASIC - Choose this option if you wish to load Spectrum 48 software (or wish to use the +2 as a 48K Spectrum).

How to choose an option

Notice that the menu option 'Tape Loader' appears to be highlighted by a 'bar'. This means that the Tape Loader option is ready to be selected - (the selection hasn't been confirmed yet). For the purpose of this example, let's assume that you *don't* want to select Tape Loader, but that instead, you want to select 128 BASIC. This means that you need to move the highlight bar to the option '128 BASIC'. To do this, use the *cursor* keys (shown below) until the highlight bar moves to the desired position.

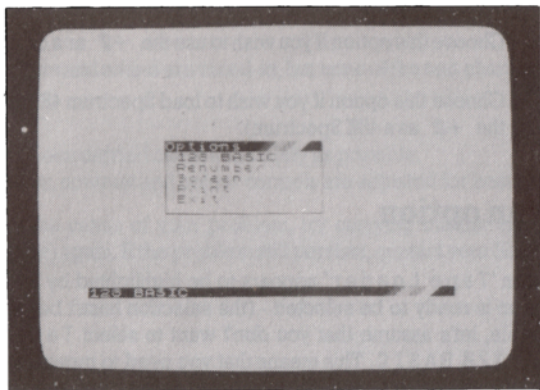


Cursor Keys

When the highlight bar is on 128 BASIC, confirm this choice by pressing the [ENTER] key.

The **+2** then switches to the 128 BASIC mode. (You will see a black horizontal 'banner' towards the bottom of the screen and a flashing *cursor* at the top left hand corner.)

Don't worry if you know nothing about BASIC - we're not going to do any programming just yet - we'll simply return to the opening menu again. To do this, we use a different menu - this one's called the 'edit menu'. Call up the edit menu by pressing the **[EDIT]** key.



Again, using the cursor keys and **[ENTER]**, select the option 'E x i t' to return to the opening menu.

You may now select whichever opening menu option you require. Depending upon your selection, refer to the following chapters for further information...

- | | |
|-------------|------------------------------------|
| Tape Loader | - Refer to chapter 3. |
| 128 BASIC | - Refer to chapters 5, 6 and 8. |
| Calculator | - Refer to chapter 9. |
| 48 BASIC | - Refer to chapters 4, 5, 7 and 8. |

IMPORTANT - Whenever you have finished using the **+2**, *always* disconnect the power supply unit from the mains supply socket.

Chapter 3

How to load Spectrum 128 software

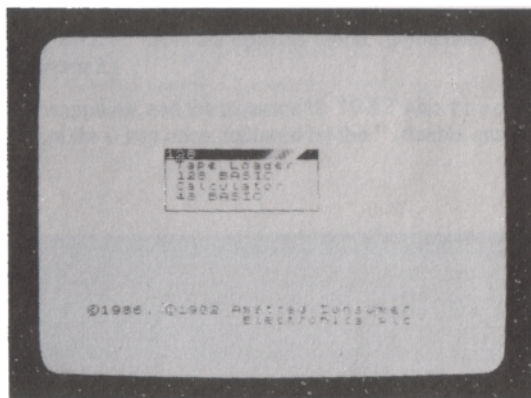
Subjects covered...

Loading software
Abandoning loading
Resetting the +2

BEWARE OF ANY SOFTWARE WHICH DOES NOT BEAR THE 'SINCLAIR QUALITY CONTROL' LOGO - For further information, read the 'Introduction' at the beginning of this manual.

To load Spectrum 128 software (a game, an utility program, etc.) carry out the following instructions...

1. Set up and switch on the +2 system, so that the opening menu appears on the screen...



2. Select the option 'Tape Loader' from the opening menu. (If you don't know how to select a menu option, refer back to chapter 2.)

3. Insert the software cassette into the datacorder and make sure that the tape is rewound to the beginning.

4. Play the cassette. As loading commences, the border colour will flash and appear striped, indicating that the program is being 'read' from the cassette. If your TV's volume control is turned up, you will also hear a varying high-pitched tone. Again, this is an indication that the program is being read.

(Note that if you wish to abandon loading, you should hold down the **[BREAK]** key until the +2 returns to the opening menu.)

Most commercially available software cassettes take a few minutes to load. Initially, the `Program:` name will appear towards the top left corner of the screen, followed by various other displays or messages (these will differ from program to program).

When the program is loaded, stop the cassette. The software is then ready to use.

If you have finished using the program and you wish to use the `+2` for something else, press and release the **RESET** button (at the left side of the `+2`). Always remember that whenever the **RESET** button is pressed, *everything* in the computer's memory (RAM) is cleared. You should therefore always make sure that you have *completely finished* with any program in the `+2`'s memory, *before* you press the button.

Chapter 4

How to load Spectrum 48 software

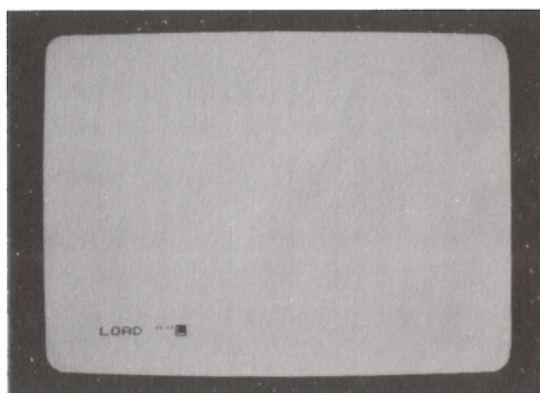
Subjects covered...

Loading software
Abandoning loading
Resetting the +2

BEWARE OF ANY SOFTWARE WHICH DOES NOT BEAR THE 'SINCLAIR QUALITY CONTROL' LOGO - For further information, read the 'Introduction' at the beginning of this manual.

To load Spectrum 48 software (a game, an utility program, etc.) carry out the following instructions...

1. Set up and switch on the +2 system, so that the opening menu appears on the screen.
2. Select the option '48 BASIC' from the opening menu. (If you don't know how to select a menu option, refer back to chapter 2.)
3. The opening menu disappears, and the message '© 1982 Amstrad' is displayed at the bottom of the screen. Now press the J key once, followed by the " (double quotes) key twice. The screen should look like this...



(If the screen does *not* correspond to the above picture, then you may have selected the wrong menu option or pressed the wrong key. In this case, press and release the **RESET** button (at the left side of the +2) and carry out steps 2 and 3 again.)

When you see the above message, press **[ENTER]**.

4. Insert the software cassette into the datacorder and make sure that the tape is rewound to the beginning.

5. Play the cassette. As loading commences, the border colour will flash and appear striped, indicating that the program is being 'read' from the cassette. If your TV's volume control is turned up, you will also hear a varying high-pitched tone. Again, this is an indication that the program is being read.

(Note that if you wish to abandon loading, you should hold down the **[BREAK]** key until the screen clears - you will then be returned to the '48 BASIC' mode. If you wish to return to the opening menu, simply press and release the **RESET** button.)

Most commercially available software cassettes take a few minutes to load. Initially, the Program : name will appear towards the top left corner of the screen, followed by various other displays or messages (these will differ from program to program).

When the program is loaded, stop the cassette. The software is then ready to use.

If you have finished using the program and you wish to use the **+2** for something else, press and release the **RESET** button. Always remember that whenever the **RESET** button is pressed, *everything* in the computer's memory (RAM) is cleared. You should therefore always make sure that you have *completely finished* with any program in the **+2**'s memory, *before* you press the button.

Chapter 5

Introduction to BASIC

The +2 uses a computer language called *BASIC* (Beginners' All-purpose Symbolic Instruction Code). BASIC is by far the commonest language for home computers, however, each type of computer tends to have its own dialect and the +2 is no exception. *Spectrum BASIC* has been designed to be easy to learn and use, though it is different from other BASICs in many respects. A complete guide to BASIC on the +2 is provided in chapter 8. If you're new to programming, however, then you should read chapter 6 (Using 128 BASIC) first. (Even if you are a seasoned BASIC user on another computer, you may still wish to read chapter 6, which describes the *editor* and other unique aspects of the +2.)

If you are used to the 48K Spectrum, then much of what is contained in this manual will no doubt seem familiar. In fact, there is a mode in which the +2 operates exactly like the old-style Spectrum - even in the editing and programming aspects. This mode isn't recommended for anything other than a history lesson for the curious; however, we have provided the relevant information (should you feel so inclined) in chapter 7 (Using 48 BASIC).

Chapter 6

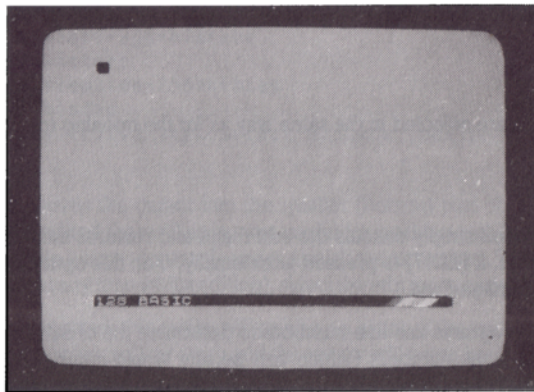
Using 128 BASIC

Subjects covered...

- The editor
- The edit menu
- Renumbering a BASIC program
- Swapping screens
- Listing to the printer
- Typing in a program
- Moving the cursor
- Running a program
- Commands and instructions

The +2 has an advanced editor to create, modify and run 128K BASIC programs. To enter the editor, select the option '128 BASIC' from the opening menu, using the cursor keys and [ENTER]. (If you don't know how to select a menu option, refer back to chapter 2.)

The screen should now look like this...



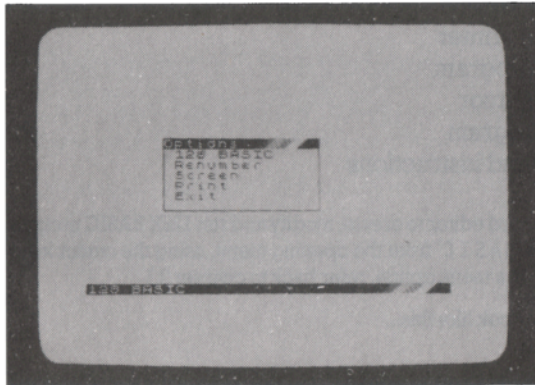
There are three things to notice about this screen.

Firstly, there is a flashing blue and white blob in the top left hand corner. This is called the *cursor*, and if you type any letters at the keyboard, then they will appear on the screen at the position of the cursor.

Secondly, there's a black bar towards the bottom of the screen. This is called the *footer bar*, and tells you which part of the +2's built-in software you're using. At the moment, it says '128 BASIC' because that's the name of the editor.

The last item of note at the moment is the small screen. This fits between the footer bar and the bottom of the screen, and is currently blank. It only has room for two lines of text, and is most often used by the +2 when it detects an error and needs to print a report to say so. It does have other uses, however, and these will be described later.

Now press the [EDIT] key. You will notice two things happen - the cursor vanishes, and a new menu appears. This is called the *edit menu*...



The edit menu's options are selected in the same way as for the opening menu (by using the cursor keys and [ENTER]).

Taking the options in turn...

128 BASIC - This option simply cancels the edit menu and restores the cursor. On the face of it - not very useful; however, if [EDIT] is pressed accidentally, then this option allows you to return to your program with no damage done.

Renumber - BASIC programs use line numbers to determine the order of the instructions to be carried out. You enter these numbers (which can be any whole-number from 1 to 9999) at the beginning of each program line you type in. Selecting the 'Renumber' option causes the BASIC program's line numbers to start at line 10 and go up in steps of 10. BASIC commands which include references to line numbers (such as GO TO, GO SUB, LINE, RESTORE, RUN and LIST) also have these references renumbered accordingly.

If for any reason it's not possible to renumber, perhaps because there's no program in the +2, or because 'Renumber' would generate line numbers greater than 9999, then the +2 makes a low-pitched bleep and the menu goes away.

Newcomers to BASIC should now skip to the section ahead which starts, 'Screen'.

It is possible, using advanced techniques, to make 'Renumber' work with values other than *start*=10 and *step size*=10. This would be useful, say, if you wanted to renumber a program containing more than 1000 lines (which could *not* be legally renumbered with line intervals of 10). The following command can be used to perform this function...

(Note - Unless you are experienced in Spectrum BASIC, you will probably not understand how this command works.)

```
LET start=5: LET stepsize=2: LET histart=INT (start/256):  
LET histep=INT (stepsize/256): POKE 23444,start-256*histart:  
POKE 23445,histart: POKE 23446,stepsize-256*histep:  
POKE 23447,histep
```

By changing the values of *start* and *stepsize*, the 'Renumber' option will renumber to any (legal) line and step size. Type in the above command, then use the option from the menu.

Later, when you have learned to write BASIC programs and save them using the datacoder, you may wish to incorporate the above into a short program for future use, for example...

```
10 INPUT "Start line", start  
20 INPUT "Step size", stepsize  
30 LET histart=INT (start/256)  
40 LET histep=INT (stepsize/256)  
50 POKE 23444,start-256*histart  
60 POKE 23445,histart  
70 POKE 23446,stepsize-256*histep  
80 POKE 23447,histep  
90 PRINT "Press [EDIT] then select Renumber option"
```

S c r e e n - This option moves the cursor into the smaller (bottom) part of the screen, and allows BASIC to be entered and edited there. This is most useful for working with graphics (see chapter 8 part 17), as any editing in the bottom screen does not disturb the top screen. To switch back to the top screen (which you can do at any time whilst editing), select the edit menu option 'S c r e e n' again.

P r i n t - If a printer is connected, this option will print-out a listing of the current program to it. When the listing has finished, the menu will go away and the cursor will come back. If for some reason the computer *cannot* print (eg. the printer is not connected or is *off-line*), then pressing the [BREAK] key twice will return you to the editor.

E x i t - This option returns you to the opening menu - the +2 retains any program that you were working on in the memory. If you wish to go back to the program again, select the option '1 2 8 B A S I C' from the opening menu.

If you select the opening menu option '4 8 B A S I C' (or if you switch off or RESET the +2) then any program in the memory will be lost. (You may, however, use the opening menu option 'C a l c u l a t o r' without losing a program in the memory.)

Reset the +2 and select '128 BASIC'. Now type in the line below. As you type it in, the characters will appear on the screen (a *character* is a letter, number, space, etc.). If you don't know how to type in the equals sign = then hold down the [SYMB SHIFT] key, then press the L key once.

Try typing in the line now...

```
10 for f=1 to 255 step 10
```

...then press [ENTER]. Providing you have spelt everything correctly, the +2 should have reprinted the line with the words FOR, TO and STEP in capital (UPPER CASE) letters, like this...

```
10 FOR f=1 TO 255 STEP 10
```

The +2 should have also emitted a short bleep, and moved the cursor to the start of the next line.

If the line remains in small (lower case) letters and you hear a low-pitched bleep, then this indicates that you have typed in something wrong. Note also that the colour of the cursor changes to red when a mistake is detected, and you *must* correct the line before it will be accepted by the +2. To do this, use the cursor keys to move to the part of the line that you wish to correct, then type in any characters you wish to insert, or use the [DELETE] key to remove any characters you wish to get rid of. When you have finally corrected the line, press [ENTER].

Now type in the line below...

(The colon : is obtained by [SYMB SHIFT] and Z, and the minus sign - is obtained by [SYMB SHIFT] and J.)

```
20 plot 0,0:draw f,175:plot 255,0:draw -f,175 (press [ENTER])
```

Don't worry about the line 'spilling over' onto the next line of the screen - the computer will take care of this and align the text so that it is easier to read. Unlike a typewriter, there's no need for you to do anything when you approach the end of a screen line because the +2 detects this automatically and moves the cursor to the beginning of a new line.

The final line of this program to type in is...

```
30 next f (press [ENTER])
```

The numbers at the beginning of each line are called *line numbers* and are used to identify each line. The line you just typed in is line 30, and the cursor should be just below, and to the left of line 30 now. Press the cursor up \uparrow key once. The cursor will move up to line 30. It doesn't move straight up, as you might expect, as there's nothing for it to move to directly above. Instead it tries to decide what you want to do, and positions itself accordingly. The cursor tries very hard to avoid blank spaces (although it doesn't mind real spaces between words on a line), and will always try and find some text to go to.

Press cursor up \uparrow once again. Now move it right (using the cursor right \rightarrow key) until it's over the 1 in DRAW - f , 175. What do you think will happen to the cursor (given its 'fear' of blank spaces) when you try to move the cursor down by one line? Try it (using the cursor down \downarrow key). As you might have expected, the cursor jumped over to the nearest text available, which in this case was at the end of line

30. Now press cursor up \uparrow again. You might have thought (since there was text directly above) that the cursor would have moved straight up - but no, it jumps back to the previous position. This is the **+2** being clever again - it realises that you haven't moved the cursor about on line 30 at all, and so remembers the last position where the cursor was actually in some text. To demonstrate this, move the cursor down again, then move it left (onto the **f**), then right, and then up, the computer thinks that you've been working on line 30 and it's therefore safe to 'forget' where you were on line 20. So the cursor moves straight up.

This sort of cursor movement is called *tracking*, and can be a little confusing at first. However, it makes editing programs much easier once it becomes familiar.

Now press **[ENTER]**. The computer opens up a new line in preparation for some new text. Type...

```
run      (press [ENTER])
```

Lots of things happen. Firstly, the footer bar and the program lines are cleared off the screen, as the 128 BASIC editor prepares to hand over control to the program you've just typed in. Then the program starts, draws a pretty pattern, and stops with the report...

```
0 OK, 30:1
```

Don't worry about what this report means.

Press **[ENTER]**. The screen will clear and the footer bar will come back, as will the program listing. This takes about a second or so, during which time the **+2** won't be taking input from the keyboard, so don't try and type anything while it's all happening.

You've just done most of the major operations necessary to program and use a computer! First, you've given the **+2** a list of instructions. *Instructions* tell the **+2** what to do (like the instruction **30 NEXT f**). Instructions have a line number and are 'stored away' rather than used immediately you type them in. Then you gave the **+2** the command **RUN** to execute the stored program.

Commands are just like instructions, only they don't have line numbers and the **+2** carries them out immediately, as soon as **[ENTER]** is pressed. In general, any instruction can be used as a command, and vice versa. It all depends on the circumstances. Every instruction or command must have at least one keyword. *Keywords* make up the vocabulary of the computer, and many of them require *parameters*. In the command **DRAW 40,200**, for example, **DRAW** is the keyword, while **40** and **200** are the parameters (telling the computer exactly *where* to do the drawing). Everything the computer does in BASIC will follow these rules.

Now press **[EDIT]** and select the **S c r e e n** option. The editor moves the program down into the bottom screen, and gets rid of the footer bar. You can only see line 10 of the program as the rest is 'hiding' off-screen (you can prove this by moving the cursor up and down).

Press **[ENTER]** then type...

```
run      (press [ENTER])
```

...and the program will run exactly the same as before. But this time, if you press **[ENTER]** afterwards, the screen *doesn't* clear, and you can move up and down the program listing (using the cursor keys) without

disturbing the top screen. If you press **[EDIT]** to get the edit menu, you might think that this would mess up the top screen. However, the **+2** remembers whatever's behind the edit menu and restores it when the menu is removed.

To prove that the editor really is working in the bottom screen, press **[ENTER]** and change line 10 to...

```
10 FOR f=1 TO 255 STEP 7
```

...by moving the cursor to the end of line 10 (just to the right of **STEP 10**), then pressing **[DELETE]** twice, and typing 7 (press **[ENTER]**).

Now type...

```
go to 10      (press [ENTER])
```

The keywords **go to** tell the **+2** not to clear the screen before starting the program. The modified program draws a slightly different pattern on top of the old one. You may continue editing the program to add further patterns, if you wish.

A word of warning - while editing in the bottom screen, don't try to edit instructions which are more than two screen lines long, for if the editor comes across an instruction which has its beginning or its end off-screen, it can become 'confused'. (The same is true of the top screen, but of course, the limitation there is unlikely to cause problems as the screen is so much larger.)

One thing you may notice while you're typing away is that **[CAPS SHIFT]** and the number keys used together do strange things: **[CAPS SHIFT]** with 5, 6, 7 and 8 move the cursor about, **[CAPS SHIFT]** with 1 calls up the edit menu, **[CAPS SHIFT]** with 0 deletes a character, **[CAPS SHIFT]** with 2 is equivalent to **[CAPS LOCK]**, and finally **[CAPS SHIFT]** with 9 selects graphics mode. All of these functions are available using the dedicated keys on the **+2**, and so there is no reason why you should ever want to use the above **[CAPS SHIFT]** and number key alternatives.

Once you're happy about how the editor works, go on to chapter 8. Again, actively experiment with the examples given and don't be afraid to try something different!

Chapter 7

Using 48 BASIC

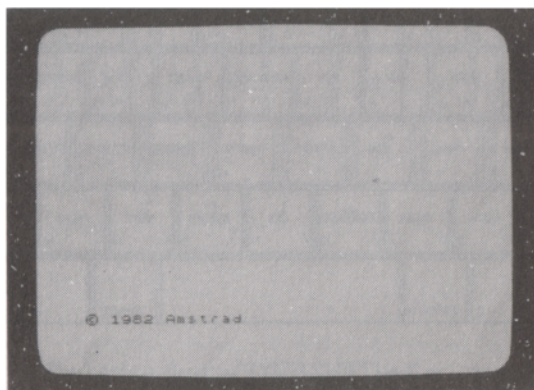
Subjects covered...

- Using the +2 as a 48K Spectrum
- Entering 48 BASIC mode
- The keyboard under 48 BASIC
- Program entry
- Editing the current line

The +2 has the ability to act exactly like a 48K Spectrum (or Spectrum +). This is achieved by selecting the '48 BASIC' mode from the opening menu. In this mode, the enhanced features of the +2 such as the extra memory, full screen editor, multi-channel sound, **RS232/MIDI** and **KEYPAD** interfaces, *cannot* be used. The **JOYSTICK 1** and **JOYSTICK 2** sockets will still operate, however.

The 48 BASIC mode is included for compatibility reasons only - there is no advantage in using 48 BASIC mode (instead of 128 BASIC mode) to write programs, and it is not recommended. The following information is included for reference only, or for anybody who is used to the 48K Spectrum and wants to use the machine immediately without having to learn about the 128 BASIC editor.

In fact, there are *two* methods to get the +2 into 48 BASIC mode; the first is by selecting the '48 BASIC' option from the opening menu (If you don't know how to select a menu option, refer back to chapter 2.) Having selected 48 BASIC, you will see the following on the screen...



The second method allows you to enter the 48 BASIC mode while editing a 128 BASIC program. To do this (while in 128 BASIC mode), type...

spectrum (press [ENTER])

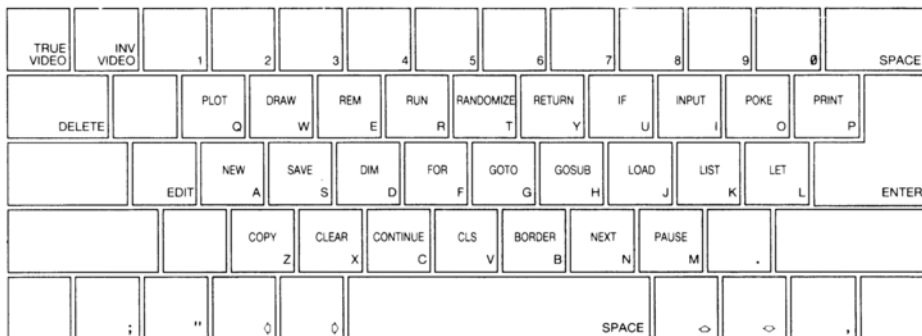
The +2 will respond with an 'OK' message. and the +2 will have changed to 48 BASIC mode, retaining any program that you had in memory. Once in 48 BASIC mode, there is no way back to 128 BASIC mode apart from resetting the +2 (or switching off, then on again).

The major difference in 48 BASIC mode is in the entering and editing of programs. The demonstration programs in chapter 8 will, in general, work in either mode, but those involving music or the 'silicon disc' *must* use 128 BASIC only. Note also that the tokens SPECTRUM and PLAY have replaced the user defined graphics characters for the keys T and U (values 163 and 164) under 128 BASIC.

Once in 48 BASIC mode, the keyboard performs as follows:

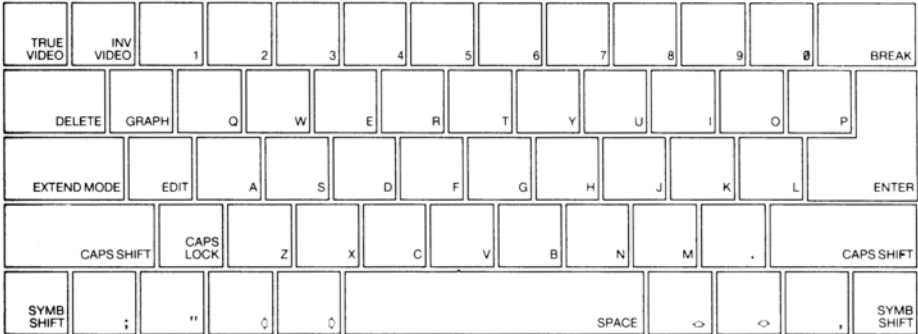
All the BASIC commands, functions and operators are available *directly* from the keyboard rather than needing to be spelled out. In order to accommodate all these functions and commands, some keys have five or more distinct meanings, obtained partly by 'shifting' the keys (ie. pressing either [CAPS SHIFT] or [SYMB SHIFT] together with the required key); and partly by having the machine in different *modes*. The flashing cursor contains a letter (K, L, C, E or G) to indicate which mode you are operating in.

K (for Keywords) mode automatically replaces L (for Letters) mode when the machine is expecting a command or program line (rather than INPUT data), and from its position on the line the +2 knows that it should expect either a line number or a keyword. K mode occurs at the beginning of a line, or after a colon : (except in a string), or after the keyword THEN. Whenever the K cursor appears, the next key pressed will be interpreted as either a keyword or a number, as follows...



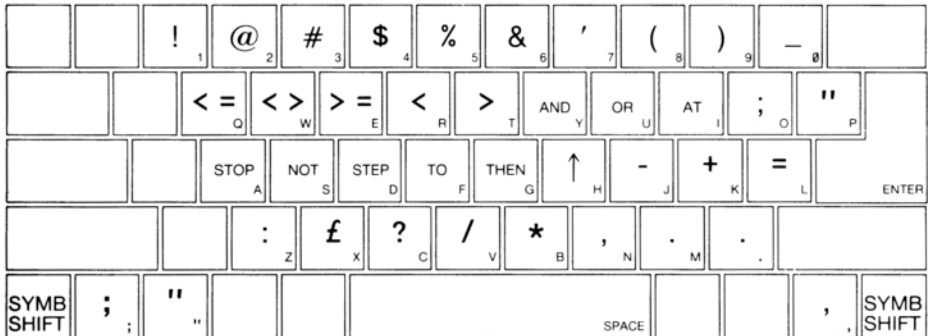
The keyboard in K mode

L (for Letters) mode normally occurs at all times (other than K mode, described above). Whenever the L cursor appears, the next key pressed will be interpreted as per the legends on the key-tops themselves, i.e...



The keyboard in L mode

In both K and L modes, pressing [SYMB SHIFT] together with a key will be interpreted as follows...



The keyboard using [SYMB SHIFT] in-K or L mode

Using **[CAPS SHIFT]** in L mode simply converts lower case letters to capitals. In K mode; however, **[CAPS SHIFT]** does not affect the keywords.

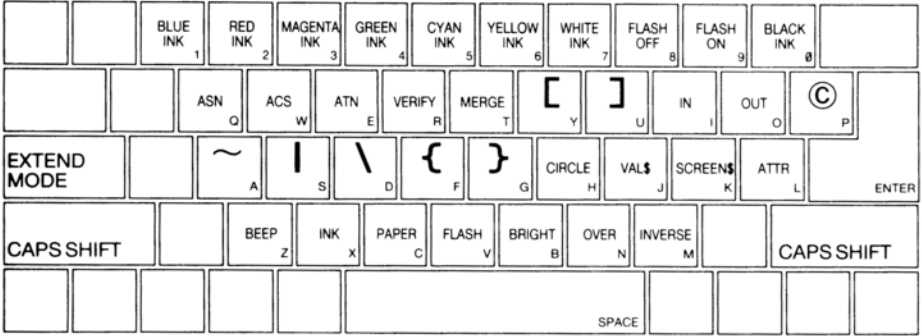
C (for Capitals) mode is a variant of L mode whereby all letters appear as capitals. The **[CAPS LOCK]** key is used to change from L mode to C mode, and back again.

E (for Extended) mode is used to obtain further characters, mostly tokens. It is entered by pressing the **[EXTEND MODE]** key, and lasts for only one character (or key depression) thereafter. Whenever the E cursor appears, the next key pressed will be interpreted as follows...

		BLUE PAPER 1	RED PAPER 2	MAGENTA PAPER 3	GREEN PAPER 4	CYAN PAPER 5	YELLOW PAPER 6	WHITE PAPER 7	BRIGHT OFF 8	BRIGHT ON 9	BLACK PAPER 0	SPACE
		SIN Q	COS W	TAN E	INT R	RND T	STR\$ Y	CHR\$ U	CODE I	PEEK O	TAB P	
EXTEND MODE		READ A	RESTORE S	DATA D	SGN F	ABS G	SQR H	VAL J	LEN K	USR L		ENTER
		LN Z	EXP X	LPRINT C	LLIST V	BIN B	INKEY\$ N	PI M				
												SPACE

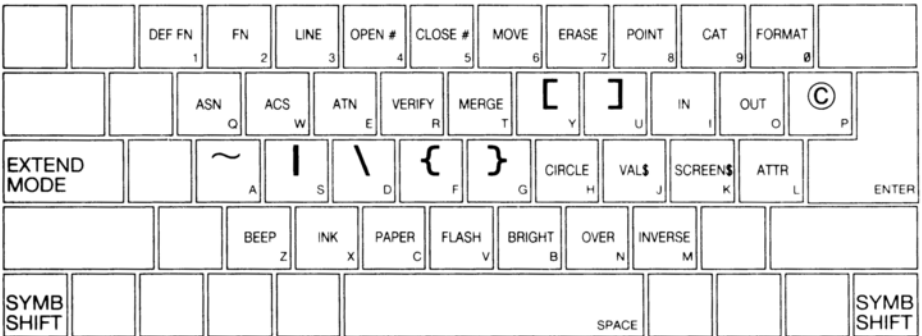
The keyboard in E mode

Applying [CAPS SHIFT] while in E mode, the next key pressed will be interpreted as follows...



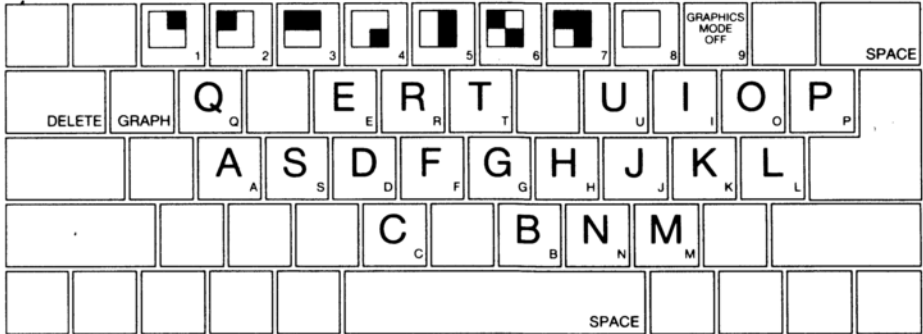
The keyboard using [CAPS SHIFT] in E mode

Applying [SYMB SHIFT] while in E mode, the next key pressed will be interpreted as follows...



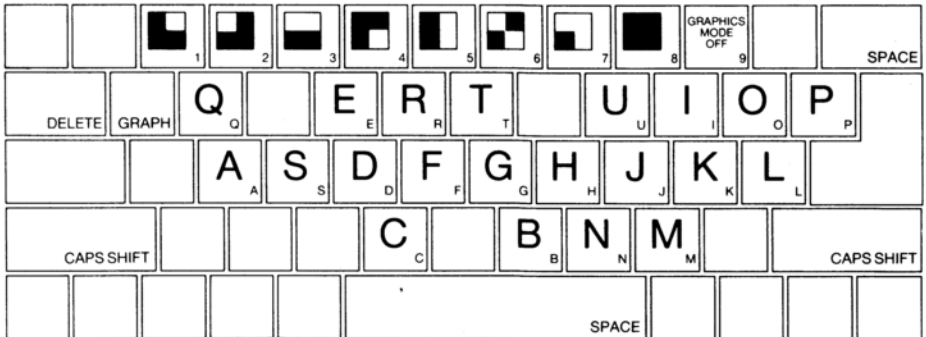
The keyboard using [SYMB SHIFT] in E mode

G (for Graphics) mode occurs when **[GRAPH]** is pressed, and lasts until it is pressed again (or **9** is pressed on its own). A number key will give a mosaic graphic, and each of the letter keys (apart from **V, W, X, Y** and **Z**) will give a user-defined graphic which, until it is defined, will look identical to an upper case character. Whenever the G cursor appears, the next key pressed will be interpreted as follows...



The keyboard in G mode

Applying **[CAPS SHIFT]** while in G mode inverts the mosaic graphics (ie. the ink colour becomes the paper colour, and the paper becomes the ink colour). Hence, the next key pressed will be interpreted as follows...



The keyboard using **[CAPS SHIFT]** in G mode

If any key is held down for more than 2 or 3 seconds, it will start repeating. Keyboard input appears in the bottom half of the screen as it is typed, each character (single symbol or compound token) being inserted just before the cursor. The cursor can be moved left and right using the cursor control keys \leftarrow and \rightarrow (to the left of the space bar). The character to the left of the cursor can be removed using **[DELETE]**.

When **[ENTER]** is pressed, the line is either executed, entered into the program, or used as INPUT data. If the line contains a *syntax error*, however, a flashing question mark ? appears next to the error.

As program lines are entered, a listing is displayed in the top half of the screen. The last line entered is called the *current line* and is indicated by the symbol > after the line number. Any line in the program may be selected as the current line (for editing purposes) by using the up and down cursor keys \uparrow and \downarrow (to the right of the space bar). To then edit the selected current line, press the **[EDIT]** key. (Editing takes place at the bottom of the screen.)

When a command is executed or a program is run, output is displayed in the top half of the screen and remains there until either **[ENTER]** or the cursor up or down key \uparrow and \downarrow is pressed. At the bottom of the screen appears a report giving a code (digit or letter) referred to in part 28 of chapter 8. This report remains on the screen until a key is pressed and the **+2** returns to K mode.

Chapter 8

A complete guide to BASIC programming

Part 1

Introduction

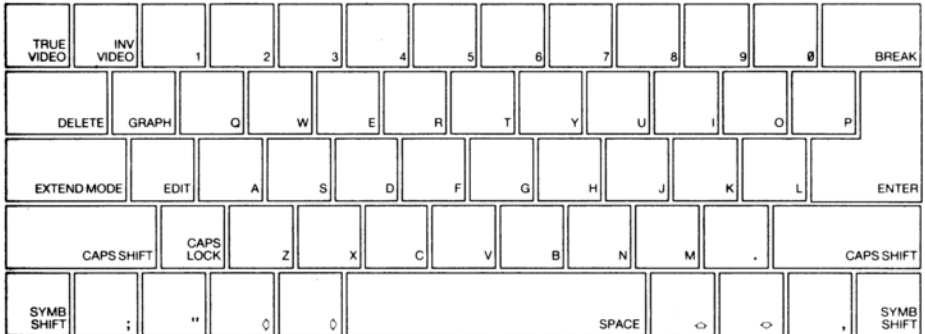
Whether you read chapter 6 first, or came straight here, you should be aware that...

Commands are obeyed straight away.

Instructions begin with a line number and are stored away for later use.

This guide to BASIC starts by repeating some things given in chapter 6 (using 128 BASIC) but in much more detail, telling you exactly what you can do. You will also find some exercises at the end of some sections - don't ignore these, as many of them illustrate points that are hinted at in the text. Look through them, and do any that interest you or that seem to cover ground that you don't understand properly. Whatever else you do, keep using your +2. If you ever wonder, 'what will it do if I type in such and such?' then the answer is simple - type it in and see! Remember, whatever you type in, it *cannot* harm the +2.

The Keyboard



The characters used on the +2 comprise not only single symbols (letters, digits, etc.) but also compound tokens (keywords, function names, etc.). Everything must be typed in full, and in most cases it doesn't matter whether capital letters (known as UPPER CASE) or small letters (lower case) are used. There are three sorts of keys on the keyboard: letter and number keys (called alphanumeric keys), symbol keys (punctuation marks), and control keys (things like [CAPS SHIFT], [DELETE] and so on).

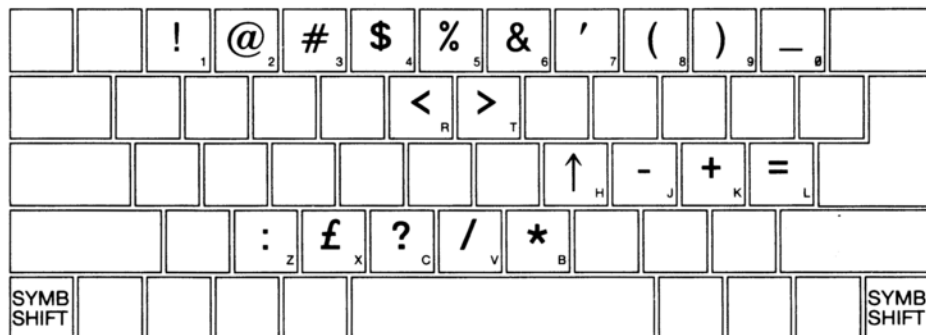
The most commonly used keys for BASIC are the alphanumeric keys. When a letter key is pressed, a lower case letter will appear on the screen together with a flashing blue and white square called the cursor. To get an upper case letter, the [CAPS SHIFT] key should be held down while the letter is typed.

If you wish to continuously type upper case letters, then pressing the [CAPS LOCK] key once will make all subsequent letters typed upper case. To return to lower case letters, simply press [CAPS LOCK] again.

To type the symbols which appear on the alphanumeric keys on the keyboard, ie...

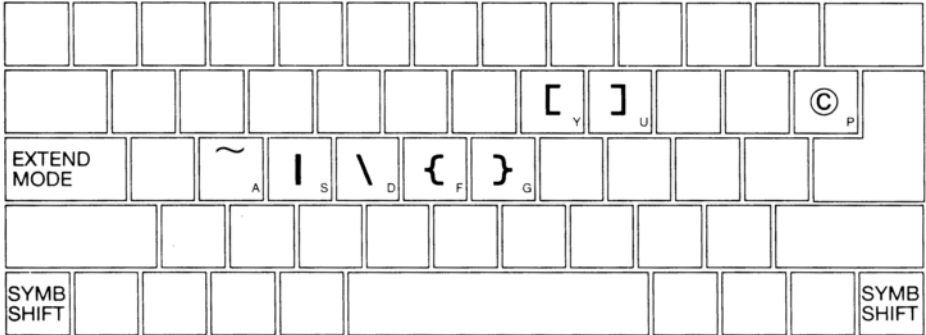
! @ # \$ % & ' () _ < > ↑ - + = : £ ? / *

...simply hold down the [SYMB SHIFT] key while the alphanumeric key with the required symbol on it is pressed (see the following diagram)...



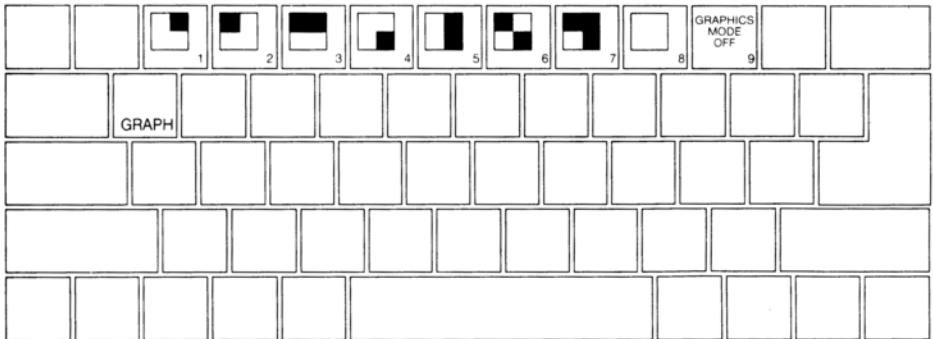
Symbols available using [SYMB SHIFT]

Additionally, the symbols [] © ~ | \ { and } can be obtained by first pressing the [EXTEND MODE] key once, then holding down [SYMB SHIFT] while pressing the appropriate alphanumeric key (see the following diagram)...



Symbols available using [SYMB SHIFT] in [EXTEND MODE]

To enter graphics mode, the [GRAPH] key is pressed once. Mosaic graphics (see the following diagram) can then be produced by pressing the number keys (except 9 and 0). [CAPS SHIFT] and the number keys produce inverted mosaic graphics. Pressing the letter keys (except T, U, V, W, X, Y and Z) produce user-defined graphics.



Mosaic graphics available using [GRAPH]

If any key is held down for more than 2 or 3 seconds, it will start repeating. As keys are pressed, a line will be built up on the screen. A line, by the way, means a line of BASIC, and may easily be several lines long on the screen. The cursor keys \leftarrow \rightarrow \uparrow \downarrow can be used to move about the line, and if the part of the line that the cursor is moved to is off screen, then the text on screen will scroll up or down to display it. Any characters typed will be inserted at the cursor, and pressing [DELETE] causes the character to the left of the cursor to be removed. As soon as [ENTER] is pressed or any attempt is made to move the cursor off the line, the +2 checks to see if the line makes sense. If it does, then there is a high-pitched bleep, and the line is either acted upon immediately or stored away as part of a program. If the line contains an error, then the +2 generates a low-pitched bleep and moves the cursor to the area where it thinks the error is (the colour of the cursor also changes to red to indicate the error). It is impossible to move off a line which contains an error - the +2 will always move the cursor back.

The monitor screen

This has 24 lines (each being 32 characters long) and is divided into two parts. The larger (top) part of the screen is at most 22 lines and displays either a listing or program output. It is the one used most often for editing. When printing in the top part has reached its bottom limit, the contents scroll up by one line. If, however, scrolling would mean losing a line that you haven't yet had a chance to see, then the **+2** stops with the message...

```
scroll?
```

Pressing any key (except **N**, **[BREAK]** or the space bar) will let scrolling continue.

Pressing one of the keys **N**, **[BREAK]** or the space bar will make the program stop with the report...

```
D BREAK - CONT repeats
```

The smaller (bottom) part of the screen is used for editing short programs, entering input data, entering direct commands (where the main screen must not be used, eg. graphics programs), and also for displaying reports.

Program entry

If the program being entered gets bigger than the screen size, the **+2** attempts to display the area of most interest (usually the last line entered together with its surrounding lines). You may, however, specify a different area of the program to be displayed using the command...

```
LIST xxx
```

...where 'xxx' is a line number, telling the **+2** to bring a specified area of the program into view.

When a command is executed or a program is run, output is displayed in the top part of the screen and remains there when the program finishes (until a key is pressed). If the program is being edited in the bottom part of the screen, then any output in the top screen will stay there until it is either overwritten, scrolled off, or a **C L S** command is issued. The bottom screen displays a report giving a code (digit or letter) referred to in part 28 of this chapter. This report remains in the bottom screen until a key is pressed.

While the **+2** is running a BASIC program, the **[BREAK]** key is checked every so often. This happens at the end of a statement, during cassette or printer use, or while music is being played.

If the **+2** finds that the **[BREAK]** key is pressed, then program execution stops, with the report...

```
D ...or... L
```

...and the program may then be edited.


Part 2

Simple programming concepts

Subjects covered...

Programs

Line numbers

Editing programs using 

RUN, LIST

GO TO, CONTINUE, INPUT, NEW, REM, PRINT

Stopping a program

Type in the first two lines of a program which will eventually print out the sum of two numbers...

```
20 print a      (press [ENTER])
10 let a=10     (press [ENTER])
```

Note that the screen looks like this...

```
10 LET a=10
20 PRINT a
```

As we have already discussed - because these lines began with numbers, they were not obeyed immediately but were stored away as program lines. You will have also noticed here that the line numbers govern the *order* in which the program lines are to be executed, and as you can see on the screen, the **+2** sorts all the lines into order whenever a new line is entered.

Note also that although we typed in each line in lower case letters, the *keywords* (ie. PRINT and LET) were converted to upper case as soon the line was entered and accepted by the **+2**. From now on, we will show information to be typed in upper case letters; however, you may continue to type in lower case letters.

So far you have only entered one number, so type...

```
15 LET b=15     (press [ENTER])
```

Now you need to change line 20 to...

```
20 PRINT a+b
```

You could type out the replacement line in full, but it is far easier to move the cursor (using the cursor keys) to just after the a, and then type...

```
+b              (don't press [ENTER] yet)
```

The line should then read...

```
20 PRINT a+b
```

Now press **[ENTER]** and the cursor will move to the line below, so that the screen looks like this...

```
10 LET a=10
15 LET b=15
20 PRINT a+b
```

Run this program by typing...

```
RUN (press [ENTER])
```

...and the sum will be displayed.

Run the program again and then type...

```
PRINT a,b (press [ENTER])
```

See how the variables are still there, even though the program has finished.

If you enter a line by mistake, say...

```
12 LET b=8
```

...and you wish to delete the line, then simply type...

```
12 (press [ENTER])
```

Line 12 will vanish, the cursor will reappear where line 12 used to be.

Now type...

```
30 (press [ENTER])
```

The **+2** will search for line 30, and since there isn't one, it will 'fall off' the end of the program. The cursor will be positioned just after the last line. If you enter any non-existent line number, then the **+2** will place the cursor where it thinks the line would have been if it really existed. This can be a useful way of moving about large programs, but **beware** - it can also be very dangerous because if the line really *did* exist before you entered the line number - it certainly *wouldn't* exist afterwards!

To list a program on the screen, simply type...

```
LIST (press [ENTER])
```

You may (particularly when working with more lengthy programs) wish to list from a certain point onwards. This can be achieved by typing an appropriate line number after the **LIST** command.

Type...

LIST 15 (press [ENTER])

...to see this illustrated.

When we were developing the above program, note how we were able to insert line 15 between the other two lines - this would have been impossible if they had been numbered 1 and 2 instead of 10 and 20. It is always good practice, therefore, to leave gaps between line numbers.

(Note that line numbers must be entered as whole numbers between 1 and 9999.)

If, at some time, you find that you haven't left enough space between line numbers, then you may use the edit menu to renumber a program. To do this, press the [EDIT] key then select the 'R e n u m b e r' option from the menu that appears; this sets the gap between each line number to 10. Try this out and see how the line numbers change.

We are now going to use the BASIC command NEW. This erases any existing programs and variables in the +2. The command should be used whenever you are about to start afresh, so type...

NEW

...and press [ENTER]. From now on, we won't mention 'press [ENTER]' every time - we'll assume that you'll remember.

With the opening menu on the screen, start up BASIC by selecting the option '1 2 8 B A S I C'.

Now carefully type in this program, which converts Fahrenheit temperatures to Celsius (centigrade)...

```
10 REM temperature conversion
20 PRINT "deg F","deg C"
30 PRINT
40 INPUT "Enter deg F",f
50 PRINT f,(f-32)*5/9
60 GO TO 40
```

Although you can type in all of line 10 in lower case, only the REM will be converted to upper case on entry as it's the only *keyword* that the +2 recognises. Also, although the words GO TO will appear with a space between them, they may be typed in as one word (GOTO) if you prefer.

Now run the program. You will see that the headings are printed on the screen (as instructed by line 20), but what has line 10 done? It looks like the +2 has completely ignored it - in fact, it has! REM in line 10 stands for remark, and is there solely to remind you of what the program does. A REM command consists of REM followed by anything you like, and the +2 will ignore everything after the REM, right up to the end of the line.

By now the +2 has got to the INPUT command in line 40 and is waiting for you to type in a value for the variable f - you can tell this because at the bottom of the screen is a flashing cursor.

Enter a number. The +2 displays the result and then waits for another number. This is because the instruction in line 60 says GO TO 40, in other words, 'instead of running out of program and stopping, jump back to line 40 and continue running from there'.

So, enter another temperature, then another...

After a few more of these you might be wondering if the machine will ever get bored with this - it won't! Next time it asks for another number, hold down [SYMB SHIFT] and type A. The word STOP will appear, and when you press [ENTER] the +2 comes back with the report...

```
H STOP in INPUT in line 40:1
```

...which tells you why it stopped, and where (in line 40). (The :1 after the line number in the report tells you that the 1st instruction in line 40 is being reported upon.)

If you wish to continue the program type...

```
CONTINUE
```

...and the +2 will ask you for another number.

When CONTINUE is used, the +2 remembers the line number in the last report that it sent you (as long as it was not 0 OK) and jumps back to that line, which in our case is line 40 (the INPUT command).

Stop the program again and replace line 60 by...

```
60 GO TO 31
```

There will be no perceptible difference to the running of the program because if the line number in a GO TO command refers to a non-existent line, then the jump is to the next line after the given number. The same goes for RUN (in fact, RUN on its own actually means RUN 0).

Now type in numbers until the screen starts getting full. When it is full, the +2 will move the whole of the top half of the screen up one line to make room, losing the heading off the top. This is called scrolling.

When you are tired of this, stop the program as before and enter the editor by pressing [ENTER].

Look at the PRINT statement in line 50. The , comma in this line is very important.

Commas are used to make the printing start either at the left-hand margin, or in the middle of the screen (depending upon which comes next). Thus in line 50, the comma causes the Celsius temperature to be printed in the middle of the line.

A semicolon ; on the other hand, is used to make the next number or string be printed immediately after the preceding one.

Another punctuation mark you can use like this in PRINT commands is the ' apostrophe. This makes whatever is printed next appear at the beginning of the next line on the screen. This also happens by default at the end of each PRINT command.

If you wish to inhibit this (so that whatever follows to be printed continues on the same line) you can put a comma or semicolon at the end of the PRINT statement. To see how this works, replace line 50 in turn by each of these...

```
50 PRINT f,  
50 PRINT f;  
50 PRINT f
```

...and run the program each time to see the difference.

The line with the comma prints everything in two columns, the line with the semicolon crams everything together, and the line without either, prints each number on a new line (you could have also used PRINT f ' to do this).

Always remember the difference between commas and semicolons in PRINT commands, and do not confuse them with : colons which are used as *separators* between commands on a single line, for example...

```
PRINT f: GO TO 40
```

Now type in these extra lines...

```
100 REM this polite program remembers your name  
110 INPUT n$  
120 PRINT "Hello ";n$;"!"  
130 GO TO 110
```

This is a separate program from the last one, but you may keep them both in the +2 at the same time. To run the new one, type...

```
RUN 100
```

Because this program expects you to input a *string* (a character or group of characters) instead of a number, it prints out two string quotes "" as a reminder. So type in a name and press [ENTER].

Next time round, you will get two string quotes again, but you don't have to use them if you don't want to. Try this, for example: rub out the quotes by pressing ⓪ twice then [DELETE] twice, and type...

```
n$
```

Since there are no string quotes, the +2 knows that it has to do some calculation - the calculation in this case is to find the value of the string variable called n\$ (which is whatever name you happen to have typed in last time round). In this way, the INPUT statement acts like LET n\$=n\$, so the value of n\$ is unchanged.

If you wish to stop the program, delete the quotes then hold down [SYMB SHIFT] and press A, then [ENTER].

Now look back at that `RUN 100` instruction which jumps to line 100 and runs the program from there. You may be asking, 'What's the difference between `RUN 100` and `GO TO 100`? Well, `RUN 100` first of all clears all the variables and the screen, and after that works just like `GO TO 100`. On the other hand, `GO TO 100` doesn't clear anything, and there may well be occasions where you wish to run a program without clearing any variables; here `GO TO` would be necessary and `RUN` could be disastrous, so it is better not to get into the habit of automatically typing `RUN` to start a program.

Another difference of course is that you may type `RUN` without a line number, and it starts off at the first line in the program. `GO TO` must *always* be followed by a line number.

Both this program and the 'temperature conversion' program stopped because you pressed **[SYMB SHIFT]** and **A** in the input line. Sometimes, by mistake, you write a program that you can't stop and that won't stop itself. Type...

```
200 GO TO 200
RUN 200
```

Although the screen is blank, the program *is* running - executing line 200 over and over again. This looks all set to go on forever unless you pull the plug out or press the reset switch! However, there is a less drastic remedy - press the **[BREAK]** key. The program will stop with the report...

```
L BREAK into program
```

At the end of every statement, the program looks to see if this key is pressed, and if it is, then the program stops. The **[BREAK]** key can also be used when you are in the middle of using the datacorder, the printer, or various other add-ons that you can attach to the +2.

In these cases there is a different report...

```
D BREAK - CONT repeats
```

The instruction `CONTINUE` in this case (and in most other cases too) repeats the statement where the program was stopped and carries straight on with the next statement (after allowing for any jumps to be made).

Run the 'name' program again and when it asks you for input, type...

```
n$ (after removing the quotes)
```

Because `n$` is an undefined variable, you will get the error report...

```
2 Variable not found
```

If you now type...

```
LET n$="fish face"
```

(which produces the report `OK, 0:1`) and then type...

CONTINUE

...you will find that you can use `n$` as input data without any trouble.

In this case `CONTINUE` does a jump to the `INPUT` command in line 110. It disregards the report from the `LET` statement because that said 'OK' and jumps to the command referred to in the previous report, ie. line 110. This feature can be extremely useful as it allows you to 'fix' a program that has stopped due to errors, and then `CONTINUE` from that point.

As we said before, the report '`L BREAK into program`' is special because after it, `CONTINUE` does not repeat the command where the program stopped.

You have now seen the statements, `PRINT`, `LET`, `INPUT`, `RUN`, `LIST`, `GO TO`, `CONTINUE`, `NEW` and `REM`, and they can all be used either as direct commands or in program lines - this is true of almost all commands in Spectrum BASIC, however, `RUN`, `LIST`, `CONTINUE` and `NEW` are not usually of much use in a program.

Exercises...

1. Put a `LIST` statement in a program, so that when you run it, it lists itself afterwards.
2. Write a program to input prices and print out the tax due (at 15 percent). Put in `PRINT` statements so that the `+2` announces what it is going to do, and asks for the input price with extravagant politeness. Modify the program so that you can also input the tax rate (to allow for zero ratings or future changes).
3. Write a program to print a running total of numbers you input. (Suggestions: have one variable called `total` - set to 0 to begin with, and another variable called `item`. Input `item`, add it to `total`, print them both, and go round again).
4. What would `CONTINUE` and `NEW` do in a program? Can you think of any uses at all for this?

Part 3

Decisions

Subjects covered...
CLS, IF, STOP
=, <, >, <=, >=, <>

All the programs we have seen so far have been pretty predictable - they went straight through the instructions, and then went back to the beginning again. This is not very useful, as in practice, we would want the +2 to make decisions and act accordingly. The instruction to do this in BASIC takes the form: 'IF something is true (or not true) THEN do something else'.

Let's look at an example of this. Use NEW to clear the previous program from the +2, select '128 BASIC', then type in and run this program. (This is clearly meant for two people to play!)

```
10 REM Guess the number
20 INPUT "Enter a secret number",a: CLS
30 INPUT "Guess the number",b
40 IF b=a THEN PRINT "That is correct": STOP
50 IF b<a THEN PRINT "That is too small, try again"
60 IF b>a THEN PRINT "That is too big, try again"
70 GO TO 30
```

Note that the CLS command (in line 20) means *clear screen*. We have used it in this program to stop the other person seeing the secret number after it is entered.

You can see that the IF statement takes the form...

IF *condition* THEN *xxx*

...where '*xxx*' stands for a command (or a sequence of commands separated by colons). The condition is something that is going to be worked out as either *true* or *false* - if it comes out as *true* then the statements in the rest of the line (after THEN) are executed; otherwise they are skipped over, and the program executes the next instruction.

The simplest conditions compare two numbers or two strings; they can test whether two numbers are equal or whether one is bigger than the other. They can also test whether two strings are equal, or whether one comes before the other in alphanumerical order. They use the symbols =, <, >, <=, >=, and <> (these are known as *relational operators*).

= means *is equal to*.
< means *is less than*.
> means *is greater than*.
<= means *is less than or equal to*.
>= means *is greater than or equal to*.
<> means *is not equal to*.

(If you keep getting mixed up about the meanings of < and >, it may help you to remember that the *thin* end of the symbol points to the number which is supposed to be *smaller*.)

In the program we have just typed in, line 40 compares a and b. If they are equal, then the program is halted by the STOP command. The report at the bottom of the screen...

```
9 STOP statement, 40:3
```

...shows that the 3rd statement (ie. the STOP command) in line 40 caused the program to halt.

Line 50 determines whether b is less than a, and line 60 whether b is greater than a. If one of these conditions is true then the appropriate comment is printed, and the program works its way down to line 70 which jumps back to line 30 and starts all over again.

Finally, note that in some versions of BASIC (not Spectrum BASIC) the IF statement can have the form...

```
IF condition THEN line number
```

This means the same as...

```
IF condition THEN GO TO line number
```

...in Spectrum BASIC.

Exercise...

1. Try this program...

```
10 LET a=1
20 LET b=1
30 IF a>b THEN PRINT a;" is higher"
40 IF a<b THEN PRINT b;" is higher"
```

Before you run it, try to work out what will be printed on the screen.

Part 4

Looping

Subjects covered...

FOR, NEXT
TO, STEP

Suppose you wish to input five numbers and add them together.

One way (don't type this in unless you are feeling dutiful) is as follows...

```
10 LET total=0
20 INPUT a
30 LET total=total+a
40 INPUT a
50 LET total=total+a
60 INPUT a
70 LET total=total+a
80 INPUT a
90 LET total=total+a
100 INPUT a
110 LET total=total+a
120 PRINT total
```

This method is not good programming practice. It may be just about controllable for five numbers, but you can imagine how tedious a program like this to add ten numbers would be, and to add a hundred would be out of the question.

Much better is to set up a variable to count up to 5 and then stop the program, like this (which you should type in)...

```
10 LET total=0
20 LET count=1
30 INPUT a
40 REM count is number of times that a has been input so far
50 LET total=total+a
60 LET count=count+1
70 IF count<=5 THEN GO TO 30
80 PRINT total
```

Notice how easy it would be to change line 70 so that this program adds ten numbers, or even a hundred.

This sort of thing is so useful that there are two special commands to make it easier - the FOR command and the NEXT command. They are always used together. Using these, the program you have just typed in does exactly the same as...

```
10 LET total=0
20 FOR c=1 TO 5
30 INPUT a
40 REM c is number of times that a has been input so far
50 LET total=total+a
60 NEXT c
80 PRINT total
```

(To get this program from the previous one, you just have to edit lines 20, 40 and 60, then delete line 70.)

Note that we have changed *count* to *c*. This is because the *control variable* of a FOR...NEXT loop must have a single letter as its name.

The effect of this program is that *c* runs through the values 1 (the initial value), 2, 3, 4 and 5 (the limit), and for each one, lines 30, 40 and 50 are executed. Then, when *c* has finished its five values, line 80 is executed.

At this point, attempt exercise 2 (which refers to the above program), at the end of this section.

An extra subtlety to the FOR...NEXT structure is that the control variable does not *have to* go up by 1 each time - you can change this 1 to anything you like by using a STEP part in the FOR command. The most general form of a FOR command is...

```
FOR control variable = initial value TO limit STEP step
```

...where the *control variable* is a single letter, and where the *initial value*, the *limit* and the *step* are all things that the +2 can calculate as numbers - like the actual numbers themselves, or sums or the names of numeric variables. So, if you replace line 20 in the program by...

```
20 FOR c=1 TO 5 STEP 3/2
```

...this will step the control variable by the amount 3/2 each time the FOR loop is executed. Note that we could have simply said STEP 1.5, or we could have assigned the step value to a variable, say *s*, and then said STEP *s*.

With the above modification, *c* will run through the values 1, 2.5 and 4. Notice that you don't have to restrict yourself to whole numbers, and also that the control value does not have to hit the limit exactly - it carries on looping as long as it is less than or equal to the limit.

At this point, attempt exercise 3 at the end of this section (which refers to the above program).

Step values can be negative instead of positive. Try this program which prints out the numbers from 1 to 10 in reverse order. (Remember, always use the command NEW before typing in a new program).

```

10 FOR n=10 TO 1 STEP -1
20 PRINT n
30 NEXT n

```

We said before that the program carries on looping as long as the control variable is less than or equal to the limit. If you consider what that would mean in this case, you'll see that it now doesn't hold true. Hence, the rule has to be modified to say that when the step is negative, the program carries on looping as long as the control variable is greater than or equal to the limit.

At this point, attempt exercises 4 and 5 at the end of this section (which refer to the above program).

You must be careful if you are running two FOR...NEXT loops together, one inside the other. Try this program, which prints out the numbers for a complete set of six spot dominoes.

```

10 FOR m=0 TO 6
20 FOR n=0 TO m
30 PRINT m;" ":"n;" ";
40 NEXT n
50 PRINT
60 NEXT m

```

You can see that the n loop is entirely inside the m loop. This means that they are properly nested.

However, what *must* be avoided is having two FOR...NEXT loops that overlap without either being entirely inside the other, like this...

```

5 REM this program is wrong
10 FOR m=0 TO 6
20 FOR n=0 TO m
30 PRINT m;" ":"n;" ";
40 NEXT m
50 PRINT
60 NEXT n

```

Two FOR...NEXT loops must either be one inside the other, or completely separate.

Another thing to avoid is jumping into the middle of a FOR...NEXT loop from the outside. The control variable is only set up properly when its FOR statement is executed, and if you miss this out, then the NEXT statement will confuse the +2. You will probably get an error report saying NEXT without FOR ...or... Variable not found.

There is nothing to stop you using a FOR...NEXT loop in a direct command. For example, try...

```
FOR m=0 TO 10: PRINT m: NEXT m
```

You can sometimes use this as a (somewhat artificial) way of getting round the restriction that you cannot GO TO anywhere inside a command - because a command has no line number. For instance...

```
FOR m=0 TO 1 STEP 0: INPUT a: PRINT a: NEXT m
```

The step size of zero here makes the command repeat itself forever.

This sort of thing is not really recommended, because if an error crops up then you have lost the command and will have to type it in again; moreover, `CONTINUE` will not work.

Exercises...

1. Make sure you thoroughly understand that a control variable not only has a name and a value, like an ordinary variable, but also a limit, a step, and a reference to the statement after the corresponding `FOR` statement. Ensure that when the `FOR` statement is executed all this information is available (using the initial value as the first value the variable takes), and also that this information is enough for the `NEXT` statement to know by how much to increase the value, whether to jump back, and if so where to jump back to.

2. Run the third program in this section, then type...

```
PRINT c
```

Why is the answer 6, and not 5?

(Answer: The `NEXT` command in line 60 is executed five times, and each time 1 is added to `c`. The last time, `c` becomes 6; so the `NEXT` command decides not to loop back, but to carry on, `c` now being past its limit).

What happens if you put `STEP 2` at the end of line 20?

3. Change the third program so that instead of automatically adding five numbers, it asks you to input the amount of numbers you wish to add. When you run this program, what happens if you input 0 (meaning that you don't wish to add any numbers)? Why might you expect this to cause problems for the `+2`, even though it is clear what you mean? (The `+2` has to make a search for the command `NEXT c`, which is not usually necessary.) In fact this has all been taken care of.

4. In line 10 of the fourth program in this section, change `10` to `100` and run the program. It will print the numbers from 100 down to 79 on the screen, and then say `scroll?` at the bottom. This is to give you a chance to see the numbers that are about to be scrolled off the top. If you press `N`, `[BREAK]` or the space bar, the program will stop with the report `D BREAK - CONT repeats`. If you press any other key, then it will print another 22 lines and ask you again if you wish to scroll.

5. Delete line 30 from the fourth program. When you run the new curtailed program, it will print the first number and stop with the message `OK`. If you then type...

```
NEXT n
```

...the program will go once round the loop, printing out the next number.

Part 5

Subroutines

Subjects covered...

GO SUB, RETURN

Sometimes, different parts of the program will have rather similar jobs to do, and you will find yourself typing in the same lines two or more times; however, this is not necessary. Instead, you need only type in the lines once (in what's called a *subroutine*) and then call the subroutine into action whenever you need it in the program.

To do this, you use the statements GO SUB (GO to SUBroutine) and RETURN. This takes the form...

```
GO SUB xxx
```

...where 'xxx' is the line number of the first line in the subroutine. It is just like GO TO xxx except that the +2 remembers where the GO SUB statement was, so that it can come back again after carrying out the subroutine.

(In case you are interested, the +2 does this by remembering at which point in the program the GO SUB command was issued (in other words where it should continue from afterwards) and storing this *return address* on top of a pile called the GO SUB stack.)

When the command...

```
RETURN
```

...is met (at the end of the subroutine itself), the +2 takes the top return address off the GO SUB stack, and continues from the next statement.

As an example, let's look at the number guessing program again. Retype it as follows...

```
10 REM "A rearranged guessing game"
20 INPUT "Enter a secret number",a:CLS
30 INPUT "Guess the number",b
40 IF b=a THEN PRINT "Correct":STOP
50 IF b<a THEN GO SUB 100
60 IF b>a THEN GO SUB 100
70 GO TO 30
100 PRINT "Try again"
110 RETURN
```

The GO TO 30 statement in line 70 (and the STOP statement in the next program) are very important because otherwise the programs will run on into their subroutines and cause an error (7 RETURN without GO SUB) when the RETURN statement is reached.

The following program uses a subroutine (from line 100 to 150) which prints a 'times table' corresponding to the value of *parameter* n. The command GO SUB 100 may be issued from any point in the program to call the subroutine. When the RETURN command in line 150 of the subroutine is reached, control returns to the main program, which continues running from the statement after the GO SUB call. Like GO TO, GO SUB may be typed in as one word (GOSUB).

```
10 REM times tables for 2, 5, 10 and 11
20 LET n=2: GO SUB 100
30 LET n=5: GO SUB 100
40 LET n=10: GO SUB 100
50 LET n=11: GO SUB 100
60 STOP
70 REM end of main program, start of subroutine
100 PRINT n;" times table"
110 FOR t=1 TO 9
120 PRINT t;" x ";n;" = ";t*n
130 NEXT t
140 PRINT
150 RETURN
```

One subroutine can happily call another, or even itself (a subroutine that calls itself is known as *recursive*).

Part 6

Data in programs

Subjects covered...
READ, DATA, RESTORE

In some previous programs we saw that information, or data, can be entered directly into the +2 using the INPUT statement. Sometimes this can be very tedious, especially if a lot of the data is repeated every time the program is run. You can save a lot of time by using the READ, DATA and RESTORE commands. For example...

```
10 READ a,b,c
20 PRINT a,b,c
30 DATA 1,2,3
```

A READ statement consists of READ followed by a list of the names of variables, separated by commas. It works rather like an INPUT statement, except that instead of getting you to type in the values to give to the variables, the +2 looks up the values in the DATA statement.

Each DATA statement is a list of expressions - numeric or string expressions - separated by commas. You can put them anywhere you like in a program, because the +2 ignores them except when it is doing a READ. You must imagine the expressions from all the DATA statements in the program as being put together to form one long list of expressions - the DATA list. The first time the +2 goes to READ a value, it reads the first expression from the DATA list; the next time, it reads the second; and thus as it meets successive READ statements, it works its way through the DATA list. (If it tries to read past the end of the DATA list, then it reports an error.)

Note that it's a waste of time putting DATA statements in a direct command, because READ will not find them. DATA statements *must* go in a program.

Let's see how all this works in the program you've just typed in. Line 10 tells the +2 to read three pieces of data and assign them to the variables a, b and c. Line 20 then says PRINT these variables. The DATA statement in line 30 provides the values of a, b and c for line 10 to read.

The information in DATA can be part of a FOR...NEXT loop. Type in...

```
10 FOR n=1 TO 6
20 DATA 2,4,6,8,10,12
30 READ d
40 PRINT d
50 NEXT n
```

Note from the above two programs that a DATA statement can appear *anywhere* - before or after the READ statement.

When the above program is run, the READ statement moves through the DATA list with each pass of the FOR...NEXT loop.

DATA statements may also contain string variables. For example...

```
10 READ d$
20 PRINT "The date is",d$
30 DATA "December 20th 1986"
```

The **+2** doesn't *have to* READ the DATA statements in order - it can be made to 'jump about' between DATA statements by using the RESTORE command. The form of the command is...

```
RESTORE xxx
```

...where 'xxx' is the line number of the DATA statement to be READ from. If you use the command RESTORE on its own (without a line number) the **+2** will jump to the first DATA statement in the program.

Type in and run the following program...

```
10 DATA 1,2,3,4,5
20 DATA 6,7,8,9
30 GO SUB 110
40 GO SUB 110
50 GO SUB 110
60 RESTORE 20
70 GO SUB 110
80 RESTORE
90 GO SUB 110
100 STOP
110 READ a,b,c
120 PRINT a'b'c
130 PRINT
140 RETURN
```

The command GO SUB 110 calls a subroutine which READs the next three items of DATA and then PRINTs them. Notice how the RESTORE command affects which items are read.

Delete line 60 and run this program again to see what happens.

Part 7

Expressions

Subjects covered...

Operations: +, -, *, /

Expressions, scientific notation, variable names

You have already seen some of the ways in which the **+2** can calculate with numbers. It can perform the four arithmetic operations +, -, * and / (remember that * is used for multiplication, and / is used for division), and it can find the value of a variable, given its name.

The example...

```
LET tax=sum*15/100
```

...illustrates that calculations can be combined. Such a combination, like `sum*15/100`, is called an expression - so an expression is just a short-hand way of telling the **+2** to do several calculations, one after the other. In our example, the expression `sum*15/100` means 'look up the value of the variable called 'sum', multiply it by 15, and divide by 100'.

A full list of the priorities of mathematical (and logical) operations will be found in part 30 of this chapter.

In expressions containing *, /, +, -, multiplication and division are carried out first - they have a higher priority than addition and subtraction. Multiplication and division have the same priority as each other, which means that they are carried out in whichever order they appear in the expression (from left to right). The next operations to be carried out are addition and subtraction - these again have the same priority as each other and so, again, are carried out in order from left to right.

Hence in the expression $8-12/4+2*2$, the first operation to be carried out is the division $12/4$ which equals 3, so we can then represent the expression as $8-3+2*2$.

The next operation to be carried out is the multiplication $2*2$ which equals 4, so the expression then becomes $8-3+4$.

Next to be carried out is the subtraction $8-3$ which equals 5, so the expression becomes $5+4$. Finally, the addition is carried out leaving the result 9.

Try this out for yourself. Type in...

```
PRINT 8-12/4+2*2
```

You may, however, change the priority of calculations within an expression by the use of brackets. Calculations within brackets are carried out *first*, so if in the above expression, you required the *addition* $4+2$ to be carried out first, you would enclose it in brackets. To see this, type in...

```
PRINT 8-12/(4+2)*2
```

...and the result this time is 4 instead of 9.

Expressions are useful because, whenever the `+2` is expecting a number from you, you can give it an expression instead and it will work out the answer.

You can also add together strings (or string variables) in a single expression. For example...

```
10 LET a$="fish"  
20 LET b$="chips"  
30 PRINT a$;" and ";b$
```

We really ought to tell you what you can and cannot use as the names of variables. As we have already said, the name of a string variable has to be a single letter followed by `$`, and the name of the control variable in a `FOR...NEXT` loop must be a single letter; but the names of ordinary numeric variables are much freer. They can use any letters or digits as long as the first one is a letter. You can put spaces in as well to make it easier to read, but they won't count as part of the name. Also, it doesn't make any difference to the name whether you type it in capitals or lower case letters. There are some restrictions about variable names which are the same as commands, however. In general, if the variable contains a BASIC command name in it with spaces around it, then it won't be accepted.

Here are some examples of the names of variables that are allowed...

```
x  
any old thing  
t42  
this name is impractical because it is too long  
tobeornottobe  
mixed cases spaces  
MixEdCAseSspAcES
```

(Note that these last two names (mixed cases spaces and `MixEdCAseSspAcES`) are considered the same, and refer to the same variable).

The following are *not* allowed as the names of variables...

```
pi (PI is a keyword)  
2001 (it begins with a digit)  
any new thing (contains NEW within two spaces)  
to be or not to be (TO, OR and NOT are all BASIC keywords)  
3bears (begins with a digit)  
M*A*S*H (* is not a letter or a digit)  
Lloyd-Webber (- is not a letter or a digit)
```

Numerical expressions can be represented by a number and exponent. Try the following to prove the point...

```
PRINT 2.34e0  
PRINT 2.34e1  
PRINT 2.34e2
```

...and so on up to...

```
PRINT 2.34e15
```

PRINT gives only eight significant digits of a number. Try...

```
PRINT 4294967295, 4294967295-429e7
```

This proves that the computer can hold the digits of 4294967295, even though it is not prepared to display them all at once.

The **+2** uses *floating point arithmetic*, which means that it keeps separate the digits of a number (its mantissa) and the position of the point (the exponent). This is not always exact, even for whole numbers. Type...

```
PRINT 1e10+1-1e10, 1e10-1e10+1
```

Numbers are held to about nine and a half digits accuracy, so $1e10$ is too big to be held exactly right. The inaccuracy (actually about 2) is more than 1, so the numbers $1e10$ and $1e10+1$ appear to the computer to be equal.

For an even more peculiar example, type...

```
PRINT 5e9+1-5e9
```

Here the inaccuracy in $5e9$ is only about 1, and the 1 to be added on in fact gets rounded up to 2. The numbers $5e9+1$ and $5e9+2$ appear to the computer to be equal. The largest integer (whole number) that can be held completely accurately is 4,294,967,294.

The string "" with no character at all is called the empty or null string. Remember that spaces are significant and an empty string is not the same as one containing nothing but spaces. Try...

```
PRINT "Did you read "The Times" yesterday?"
```

When you press **[ENTER]** you will get the flashing red cursor that shows there is a mistake somewhere in the line. When the **+2** finds the double quotes at the beginning of "The Times" it imagines that these mark the end of the string "Did you read", and it then can't work out what The Times means.

There is a special device to get over this - whenever you wish to write a string quote symbol in the middle of a string, you must write it twice, like this...

```
PRINT "Did you read ""The Times"" yesterday?"
```

As you can see from what is printed on the screen, each double quote is only really there once - you just have to type it twice to get it recognised.

Part 8

Strings

Subjects covered...
Slicing, using T O

Given a string, a *substring* of it consists of some consecutive characters from it, taken in sequence. thus "string" is a substring of "bigger string", but "b sting" and "big reg" are not.

There is a notation called *slicing* for describing substrings, and this can be applied to arbitrary string expressions. The general form is...

string expression (start T O finish)

...so that, for instance...

" abcdef " (2 T O 5)

...is equal to bcde.

If you omit the start, then 1 is assumed; if you omit the finish, then the length of the string is assumed. Thus...

" abcdef " (T O 5) is equal to abcde
" abcdef " (2 T O) is equal to bcdef
" abcdef " (T O) is equal to abcdef

You can also write this last one as " abcdef " ().

A slightly different form misses out the T O and just has one number.

" abcdef " (3) is equal to " abcdef " (3 T O 3) is equal to c

Although normally both start and finish must refer to existing parts of the string, this rule is overridden by another one: if the start is more than the finish, then the result is the empty string. So...

" abcdef " (5 T O 7)

...gives the error 3 Subscript wrong because the string only contains 6 characters and 7 is too many, but...

" abcdef " (8 T O 7)

...and...

" abcdef " (1 T O Ø)

...are both equal to the empty string "" and are therefore permitted.

The start and finish must not be negative, or you get the error `B i n t e g e r o u t o f r a n g e`. This next program is a simple one illustrating some of these rules...

```
10 LET a$="abcdef"
20 FOR n=1 TO 6
30 PRINT a$(n TO 6)
40 NEXT n
```

Type `NEW` when this program has been run, and enter the next program.

```
10 LET a$="1234567890"
20 FOR n=1 TO 10
30 PRINT a$(n TO 10),a$((11-n) TO 10)
40 NEXT n
```

For string variables, we can not only extract substrings, but also assign to them. For instance, type...

```
LET a$="I love my Sinclair"
```

...and then...

```
LET a$(11 TO 18)="Amstrad*****"
```

...and...

```
PRINT a$
```

Notice how since the substring `a$(11 TO 18)` is only 8 characters long, only its first 8 characters (`Amstrad*`) are used; the remaining 4 characters (`****`) are discarded. This is a characteristic of assigning to substrings: the substring has to be exactly the same length afterwards as it was before. To make sure this happens, the string that is being assigned to it is cut off on the right if it is too long, or filled out with spaces if it is too short - this is called 'Procrustean assignment' after the inn-keeper Procrustes who used to make sure that his guests fitted their beds by either stretching them out on a rack or cutting their feet off!

If you now try...

```
LET a$()="Hello there"
```

...and...

```
PRINT a$;"."
```

...you will see that the same thing has happened again (this time with spaces put in) because `a$()` counts as a substring.

```
LET a$="Hello there"
```

...will do it properly.

Complicated string expressions will need brackets around them before they can be sliced. For example...

```
"abc"+"def"(1 TO 2) is equal to "abcde"  
("abc"+"def")(1 TO 2) is equal to "ab"
```

Exercise...

1. Try writing a program to print out the day of the week using string slicing. (Hint - Let the string be *SunMonTueWedThuFriSat*).

Part 9

Functions

Subjects covered...

DEF
LEN, STR\$, VAL, SGN, ABS, INT, SQR
FN

Consider the sausage machine. You put a lump of meat in at one end, turn a handle and out comes a sausage at the other end. A lump of pork gives a pork sausage, a lump of fish gives a fish sausage, and a lump of beef a beef sausage.

Functions are practically indistinguishable from sausage machines but there is a difference; they work on numbers and strings instead of meat. You supply one value (called the argument), mince it up by doing some calculations on it, and eventually get another value - the result.

Meat in → Sausage Machine → Sausage out
Argument in → Function → Result out

Different arguments give different results, and if the argument is completely inappropriate the function will stop and give an error report.

Just as you can have different machines to make different products - one for sausages, another for dish cloths, a third for fish-fingers, and so on, different functions will do different calculations. Each will have its own value to distinguish it from the others.

You use a function in expressions by typing its name followed by the argument, and when the expression is evaluated the result of the function will be worked out.

As an example, there is a function called LEN, which works out the length of a string. Its argument is the string whose length you wish to find, and its result is the length, so that if you type...

```
PRINT LEN "Spectrum +2"
```

the **+2** will write the answer 11, ie. the number of characters (including spaces) in the string 'Spectrum +2'.

If you mix functions and operations in a single expression, then the functions will be worked out before the operations. Again, however, you can circumvent this rule by using brackets. For instance, here are two expressions which differ only in the brackets, and yet calculations are performed in an entirely different order in each case (although, as it happens, the end results are the same).

```
LEN "Fred" + LEN "Bloggs"      LEN ("Fred" + "Bloggs")  
4+LEN "Bloggs"                LEN ("FredBloggs")  
4+6                              LEN "FredBloggs"  
10                               10
```

Here are some more functions...

STR\$ converts numbers into strings: its argument is a number, and its result is the string that would appear on the screen if the number were displayed by a **PRINT** statement. Note how its name ends in a **\$** sign to show that its result is a string. For example, you could say...

```
LET a$=STR$ 1e2
```

...which would have exactly the same effect as typing...

```
LET a$="100"
```

Or you could say...

```
PRINT LEN STR$ 100.0000
```

...and get the answer 3, because **STR\$ 100.0000** is equal to **100**, the length of which is 3 characters.

VAL is like **STR\$** in reverse - it converts strings into numbers. For instance...

```
VAL "3.5"
```

...is equal to the number 3.5.

VAL is the reverse of **STR\$** because if you take any number, apply **STR\$** to it, and then apply **VAL** to it, you get back to the number you first thought of.

However, if you take a string, apply **VAL** to it, and then apply **STR\$** to it, you do not always get back to your original string.

VAL is an extremely powerful function, because the string which is its argument is not restricted to looking like a plain number - it can be any numeric expression. Thus, for instance...

```
VAL "2*3"
```

...is equal to 6. Even...

```
VAL ("2"+"*3")
```

...is equal to 6. There are two processes at work here. In the first, the argument of **VAL** is evaluated as a string - the string expression **"2"+"*3"** is evaluated to give the string **"2*3"**. Then, the string has its double quotes stripped off, and what is left is evaluated as a number: so **2*3** is evaluated to give the number 6.

This can get pretty confusing if you don't keep your wits about you; for instance...

```
PRINT VAL "VAL""VAL""""2"""""""""
```

(Remember that inside a string, a string quote must be written twice. If you go down into further depths of strings, then you find that string quotes need to be quadrupled, or even octupled.)

There is another function, rather similar to VAL, though probably less useful, called VAL\$. Its argument is still a string, but its result is also a string. To see how this works, recall how VAL goes in two steps: first its argument is evaluated as a string, then the string quotes stripped off this, and whatever is left is evaluated as a number. With VAL\$, the first step is the same, but after the string quotes have been stripped off in the second step, whatever is left is evaluated as another string. Thus...

```
VAL$ ""Ursula"" is equal to "Ursula"
```

(Notice how the string quotes proliferate again.) Try...

```
LET a$="99"
```

...and print out all of the following: VAL a\$, VAL "a\$", VAL ""a\$"", VAL\$ a\$, VAL\$ "a\$" and VAL\$ ""a\$"". Some of these will work, and some of them won't; try to explain all the answers. (Keep a cool head).

SGN is the sign function (sometimes called signum). It is the first function you have seen that has nothing to do with strings, because both its argument and its result are numbers. The result is +1 if the argument is positive, 0 if the argument is zero, and -1 if the argument is negative.

ABS is another function whose argument and result are both numbers. It converts the argument into a positive number (which is the result) by forgetting the sign, so that for instance...

```
ABS -3.2
```

...is equal to

```
ABS 3.2
```

...which is simply equal to 3.2.

INT stands for 'integer part' - an integer is a whole number, possibly negative. This function converts a fractional number into an integer by throwing away the fractional part, so that for instance...

```
INT 3.9
```

...is equal to 3.

Be careful when you are applying it to negative numbers, because it always rounds down. Thus for instance...

```
INT -3.1
```

...is equal to -4.

SQR calculates the square root of a number, ie. the result that, when multiplied by itself, gives the argument, for instance...

```
SQR 4
```

...is equal to 2 because $2^2 = 4$.

```
SQR 0.25
```

...is equal to 0.5 because $0.5^2 = 0.25$.

```
SQR 2
```

...is equal to 1.4142136 (approx) because $1.4142136^2 = 2.0000001$.

If you multiply any number (even a negative one) by itself, the answer is always positive. This means that negative numbers do not have square roots, so if you apply SQR to a negative argument you get the error report `A Invalid Argument`.

You can also define functions of your own. Possible names for these are FN followed by a letter (if the result is a number) or FN followed by a letter followed by \$ (if the result is a string). These functions are much stricter about brackets - the argument *must* be enclosed in brackets.

You define a function by putting a DEF statement somewhere in the program. For instance, here is the definition of a function FN s whose result is the square of the argument...

```
10 DEF FN s(x)=x*x: REM the square of x
```

The s following the DEF FN is the name of the function. The x in brackets is a name by which you wish to refer to the argument of the function. You can use any single letter you like for this (or, if the argument is a string, a single letter followed by \$).

After the = sign comes the actual definition of the function. This can be any expression, and it can also refer to the argument using the name you've given it (in this case, x) as though it were an ordinary variable.

When you have entered this line, you can invoke the function just like one of the +2's own functions, by typing its name, FN s, followed by the argument Remember that when you have defined a function yourself, the argument must be enclosed in brackets. Try it out a few times...

```
PRINT FN s(2)
PRINT FN s(3+4)
PRINT 1+INT FN s (LEN "chicken"/2+3)
```

Once you have put the corresponding DEF statement into the program, you can use your own functions in expressions just as freely as you can use the computer's.

INT always rounds down. To round to the nearest integer, add 0.5 first - you could write your own function to do this...

```
20 DEF FN r(x)=INT (x+0.5): REM gives x rounded to
   the nearest integer.
```

You will then get, for instance...

```
FN r(2.9) is equal to 3      FN r(2.4) is equal to 2
FN r(-2.9) is equal to -3   FN r(-2.4) is equal to -2
```

Compare these with the answers you will get when you use INT instead of FN r. Type in and run the following...

```
10 LET x=0: LET y=0: LET a=10
20 DEF FN p(x,y)=a+x*y
30 DEF FN q( )=a+x*y
40 PRINT FN p(2,3),FN q( )
```

There are a lot of subtle points in this program. First, a function is not restricted to just one argument: it can have more, or even none at all - but you must still always keep the brackets.

Second, it doesn't matter whereabouts in the program you put the DEF statements. After the +2 has executed line 10, it simply skips over lines 20 and 30 to get to line 40. They do, however, have to be somewhere in the *program* - they can't be in a command.

Third, x and y are both the names of variables in the program as a whole, and the names of arguments for the function FN p. FN p temporarily forgets about the variables called x and y, but since it has no *argument* called a, it still remembers the *variable* a. Thus when FN p(2,3) is being evaluated, a has the value 10 because it is the variable, x has the value 2 because it is the first argument, and y has the value 3 because it is the second argument. The result is then, 10+2*3 which is equal to 16. When FN q() is being evaluated, on the other hand, there are no arguments, so a, x and y all still refer to the *variables* and so have the values 10, 0 and 0 respectively. The answer in this case is 10+0*0 which is equal to 10.

Now change line 20 to...

```
20 DEF FN p(x,y)=FN q( )
```

This time, FN p(2,3) will have the value 10 because FN q will still go back to the variables x and y rather than using the arguments of FN p.

Some BASICs (not Spectrum BASIC) have functions called LEFT\$, RIGHT\$, MID\$ and TL\$.

LEFT\$(a\$,n) gives the substring of a\$ consisting of the first n characters.

RIGHT\$(a\$,n) gives the substring of a\$ consisting of the characters from nth on.

MID\$(a\$, n1, n2) gives the substring of **a\$** consisting of **n2** characters, starting at the **n1**th. **TL\$(a\$)** gives the substring of **a\$** consisting of all its characters except the first.

You can write some user-defined functions to do the same...

```
10 DEF FN t$(a$)=a$(2 TO ): REM TL$
20 DEF FN l$(a$,n)=a$( TO n): REM LEFT$
```

Check that these work with strings of length 0 or 1. Note that our **FN l\$** has two arguments, one a number and the other a string. A function can have up to 26 numeric arguments (why 26?) and at the same time up to 26 string arguments.

Exercise...

Use the function **FN s(x) = x * x** to test **SQR**. You should find that...

```
FN s(SQR x)
```

...equals **x** if you substitute any positive number for **x**, and...

```
SQR FN s(x)
```

...equals **ABS x** whether **x** is positive or negative (Why is the **ABS** there?).

Part 10

Mathematical functions

Subjects covered...

↑

P I, EXP, LN, SIN, COS, TAN, ASN, ACS, ATN

This section deals with the mathematics that the **+2** can handle. Quite possibly you will never have to use any of this at all, so if you find it too heavy going, don't be afraid of skipping it. It covers the operation \uparrow (raising to a power), the functions EXP and LN, and the trigonometrical functions SIN, COS, TAN and their inverses ASN, ACS, and ATN.

\uparrow and EXP

You can raise one number to the power of another. This means 'multiply the first number by itself the second number of times'. This is normally shown by writing the second number just above and to the right of the first number; but obviously this would be difficult on a computer so we use the symbol \uparrow instead. For example, the powers of 2 are...

$$2 \uparrow 1 = 2$$

$$2 \uparrow 2 = 2 * 2 = 4 \text{ (2 squared, normally written } 2^2\text{)}$$

$$2 \uparrow 3 = 2 * 2 * 2 = 8 \text{ (2 cubed, normally written } 2^3\text{)}$$

$$2 \uparrow 4 = 2 * 2 * 2 * 2 = 16 \text{ (2 to the power of four, normally written } 2^4\text{)}$$

...and so on.

Thus, at its most elementary level, ' $a \uparrow b$ ' means 'a multiplied by itself b times', but obviously this only makes sense if b is a positive whole number. To find a definition that works for other values of b, we consider the rule...

$$a \uparrow (b+c) = a \uparrow b * a \uparrow c$$

(Notice that we give \uparrow a higher priority than * and / so that when there are several operations in one expression, the \uparrow s are evaluated before the *s and /s). You should not need much convincing that this works when b and c are both positive whole numbers; but if we decide that we want it to work even when they are not, then we find ourselves compelled to accept that...

$$a \uparrow 0 = 1$$

$$a \uparrow (-b) = 1/a \uparrow b$$

$$a \uparrow (1/b) = \text{the } b\text{th root of } a, \text{ which is to say, the number that you have to multiply by itself } b \text{ times to get } a$$

...and...

$$a \uparrow (b*c) = (a \uparrow b) \uparrow c$$

If you have never seen any of this before then don't try to remember it straight away, just remember that...

$$a \uparrow (-1) = 1/a$$

...and...

$$a \uparrow (1/2) = \text{SQR } a$$

...and maybe when you are familiar with these, the rest will begin to make sense.

Experiment with all this by trying this program...

```
10 INPUT a,b,c
20 PRINT a ↑ (b+c), a ↑ b * a ↑ c
30 GO TO 10
```

Of course, if the rule we gave earlier is true, then each time round, the two numbers that the `+2` prints out will be equal. (Note - because of the way the computer works out `↑`, the number on the left, `a` in this case, must *never* be negative.)

A rather typical example of what this function can be used for is that of compound interest. Suppose you keep some of your money in a building society and they give 15% interest per year. Then after one year you will have not just the 100% that you had anyway, but also the 15% interest that the building society has given you, making altogether 115% of what you had originally. To put it another way, you have multiplied your sum of money by 1.15, and this is true however much you had there in the first place. After another year, the same will have happened again, so that you will then have $1.15 * 1.15$, or in other words, $1.15 \uparrow 2$, or in other words, 1.3225 times your original sum of money. In general then, after y years, you will have $1.15 \uparrow y$ times what you started out with.

If you try this command...

```
FOR y=0 TO 100: PRINT y, 10 * 1.15 ↑ y: NEXT y
```

...you will see that even starting off from just £10, it all mounts up quite quickly, and what's more, it gets faster and faster as time goes on (though you might still find that it doesn't keep up with inflation).

This sort of behaviour, where after a fixed interval of time some quantity multiplies itself by a fixed proportion, is called *exponential growth*, and it is calculated by raising a fixed number to the power of the time.

Suppose you did this...

```
10 DEF FN a(x)=a ↑ x
```

Here, `a` is more or less fixed, by `LET` statements - its value will correspond to the interest rate, which changes only every so often.

There is a certain value for `a` that makes the function `FN a` look especially pretty to the trained eye of a mathematician; and this value is called *e*. The `+2` has a function called `EXP` defined by...

```
EXP x is equal to e ↑ x
```

Unfortunately, e itself is not an especially pretty number; it is an infinite non-recurring decimal. You can see its first few decimal places by typing...

```
PRINT EXP 1
```

...because $\text{EXP } 1 = e \uparrow 1 = e$. Of course, this is just an approximation. You can never write down e exactly.

LN

The inverse of an exponential function is a *logarithmic function* - the *logarithm* (to base a) of a number x is the power to which you'd have to raise a to get the number x , and this is written $\log_a x$. Thus by definition, $a \uparrow \log_a x = x$; and it is also true that $\log(a \uparrow x) = x$.

You may well already know how to use base 10 logarithms for doing multiplications; these are called common logarithms. The **+2** has a function LN which calculates logarithms to the base e ; these are called natural logarithms. To calculate logarithms to any other base, you must divide the natural logarithm by the natural logarithm of the base, ie. $\log_a x = \text{LN } x / \text{LN } a$.

PI

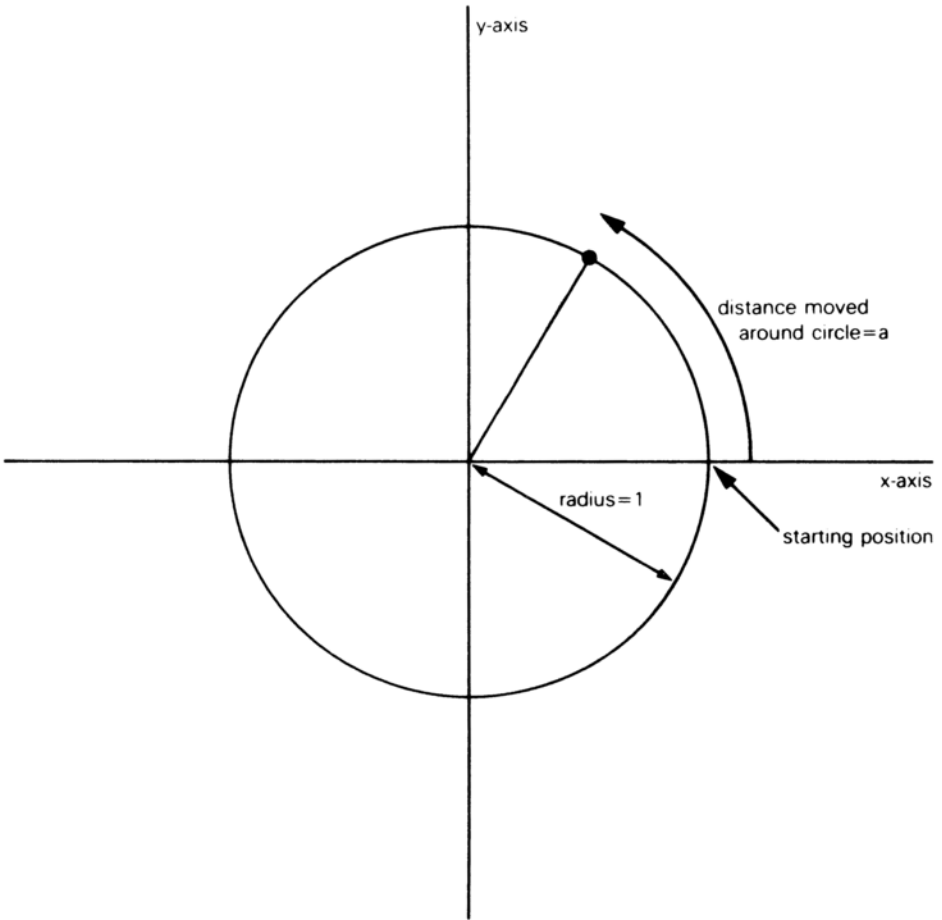
Given any circle, you can find its *perimeter* (the distance round its edge - often called its circumference) by multiplying its diameter (width) by a number called π . π is a Greek p, and it is used because it stands for perimeter. Its name is *pi*.)

Like e , π is an infinite non-recurring decimal - it starts off as 3.1415927. The word PI on the **+2** is taken as standing for this number. Try ...

```
PRINT PI
```

SIN COS and TAN, ASN ACS and ATN

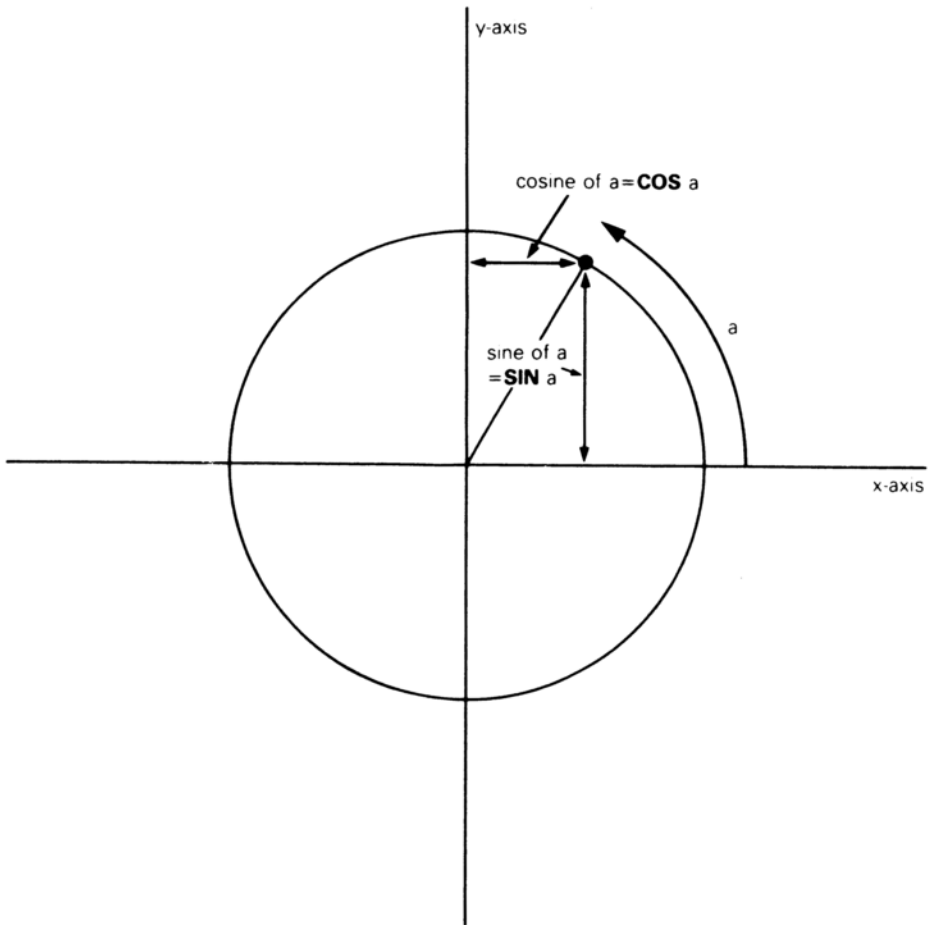
The *trigonometrical* functions measure what happens when a point moves round a circle. Here is a circle of radius 1 ('1 what?' you may ask - it doesn't matter, as long as we keep to the same unit all the way through) and a point moving round it. The point started at the '3 o'clock' position, and then moved round in an anti-clockwise direction.



We have also drawn in two lines called *axes* through the centre of the circle. The one through 3 o'clock is called the *x-axis*, and the one through 12 o'clock is called the *y-axis*.

To specify where the point is, you say how far it has moved round the circle from its 3 o'clock starting position: let us call this distance *a*. We know that the circumference of the circle is 2π (because its radius is 1 and its diameter is thus 2); so when it has moved a quarter of the way round the circle, $a = \pi/2$; when it has moved halfway round, $a = \pi$; and when it has moved the whole way round, $a = 2\pi$.

Given the curved distance round the edge - *a*, two other distances you might like to know are how far the point is to the *right of the y-axis*, and how far it is *above the x-axis*. These are called, respectively, the *cosine* and *sine* of *a*. The functions **C O S** and **S I N** on the **+2** will calculate these.



Note that if the point goes to the left of the y -axis, then the cosine becomes negative, and if the point goes below the x -axis, the sine becomes negative.

Another property is that once a has got up to 2π , the point is back where it started and the sine and cosine start taking the same values all over again, ie. $\text{SIN}(a + 2 * \text{PI})$ equals $\text{SIN } a$, and $\text{COS}(a + 2 * \text{PI})$ equals $\text{COS } a$.

The *tangent* of a is defined as being the sine divided by the cosine; the corresponding function on the $+2$ is called **TAN**.

Sometimes we need to work these functions out in reverse, finding the value of a that has given sine, cosine or tangent. The functions to do this are called arcsine (**ASN** on the $+2$), arcosine (**ACS**) and arctangent (**ATN**).

In the diagram of the point moving round the circle, look at the radius joining the centre to the point. You should be able to see that the distance we have called a (the distance that the point has moved round the edge of the circle) is a way of measuring the angle through which the radius has moved away from the x -axis. When $a = \pi/2$, the angle is 90 degrees; when $a = \pi$ the angle is 180 degrees, and so on, round to when $a = 2\pi$, and the angle is 360 degrees. You might just as well forget about degrees, and measure the angle in terms of a alone; we say then that we are measuring the angle in *radians*. Thus $\pi/2$ radians = 90 degrees and so on.

You must always remember that on the $+2$, the functions **SIN**, **COS**, etc. use radians and *not* degrees. To convert degrees to radians, divide by 180 and multiply by π ; to convert back from radians to degrees, you divide by π and multiply by 180.

Part 11

Random Numbers

Subjects covered...

RANDOMIZE
RND

This section deals with the keywords RND and RANDOMIZE.

In some ways RND is like a function - it does calculations and produces a result. It is unusual in that it does not need an argument.

Each time you use it, its result is a new random number between 0 and 1. (Sometimes it can take the value 0, but never 1.)

Try...

```
10 PRINT RND
20 GO TO 10
```

...to see how the answer varies. Can you detect any pattern? You shouldn't be able to - 'random' means that there is no pattern.

Actually, RND is not truly random, because it follows a fixed sequence of 65536 numbers. However, these are so thoroughly jumbled up that there are at least no obvious patterns, and we say that RND is *pseudo-random*.

RND gives a random number between 0 and 1, but you can easily get random numbers in other ranges. For instance, $5 * \text{RND}$ is between 0 and 5, and $1.3 + 0.7 * \text{RND}$ is between 1.3 and 2. To get whole numbers, use INT (remembering that INT always rounds down) as in $1 + \text{INT}(\text{RND} * 6)$, which we shall use in a program to simulate dice. $\text{RND} * 6$ is in the range 0 to 6, but since it never actually reaches 6, INT (RND*6) is 0, 1, 2, 3, 4 or 5.

Here is the program...

```
10 REM dice throwing program
20 CLS
30 FOR n=1 TO 2
40 PRINT 1+INT (RND*6); " ";
50 NEXT n
60 INPUT a$: GO TO 20
```

Press [ENTER] each time you wish to 'throw' the dice.

The `RANDOMIZE` statement may be used to make `RND` start off at a definite place in its sequence of numbers, as you can see with this program...

```
10 RANDOMIZE 1
20 FOR n=1 TO 5: PRINT RND,: NEXT n
30 PRINT: GO TO 10
```

After each execution of `RANDOMIZE 1`, the `RND` sequence starts off again with 0.0022735596. You can use other numbers between 1 and 65535 in the `RANDOMIZE` statement to start the `RND` sequence off at different places.

If you had a program with `RND` in it and it also had some mistakes that you had not found, then it would help to use `RANDOMIZE` like this so that the program behaved the same way each time you ran it.

`RANDOMIZE` on its own (or `RANDOMIZE 0`) have a different effect - they really *do* randomise `RND` -you can see this in the next program...

```
10 RANDOMIZE
20 PRINT RND: GO TO 10
```

The sequence you get here is not very random, because `RANDOMIZE` uses the *time* since the `+2` was switched on. As this has gone up by the same amount each time `RANDOMIZE` is executed, the next `RND` does more or less the same. You would get better 'randomness' by replacing `GO TO 10` by `GO TO 20`.

Here is a program to toss coins and count the numbers of heads and tails...

```
10 LET heads=0: LET tails=0
20 LET coin=INT (RND*2)
30 IF coin=0 THEN LET heads=heads+1
40 IF coin=1 THEN LET tails=tails+1
50 PRINT heads;" ";tails,
60 IF tails <>0 THEN PRINT heads/tails;
70 PRINT: GO TO 20
```

The ratio of heads to tails should become approximately 1 if you go on long enough, because in the long run you expect approximately equal numbers of heads and tails.

Exercise...

1. Suppose you choose a number between 1 and 872 and type...

```
RANDOMIZE your number
```

Convince yourself that the next value of `RND` will be...

```
(75*(your number+1)-1)/65536
```

Try this out for yourself.

Part 12

Arrays

Subjects covered...

Arrays (Note that the way that the **+2** handles string arrays is slightly non-standard).

D I M

Suppose that you have a list of numbers - for instance, the marks of ten people in a class. To store them in the **+2** you could use the variables *m1*, *m2*, *m3*... and so on up to *m10*, but the program to set up these ten variables would be rather long and tedious to type in, ie...

```
10 LET m1=75
20 LET m2=44
30 LET m3=90
40 LET m4=38
50 LET m5=55
60 LET m6=64
70 LET m7=70
80 LET m8=12
90 LET m9=75
100 LET m10=60
```

Instead, there is a mechanism, known as an *array* whereby you may specify a variable which (instead of containing a single value as variables normally do) may contain a number of separate *elements*, each of which may contain different values. Each element is referenced by an *index* number (the *subscript*) written in brackets after the variable name. For the above example, the array variable's name could be *m* - (the name of an array variable *must* be a single letter), and the ten variables would then be *m(1)*, *m(2)*, *m(3)*... and so on up to *m(10)*.

The elements of an array are called *subscripted variables*, as opposed to the simple variables that you are already familiar with.

Before you can use an array, you must reserve some space for it inside the **+2**, and you do this using the keyword **D I M** (for dimension). The statement...

```
D I M m(10)
```

...sets up an array called *m* whose dimensions are 10 (ie. there are 10 subscripted variables). The **D I M** statement initialises each element in the array to zero. It also deletes any array called *m* that existed previously - (however, it *doesn't* delete any *simple variable* called *m*. An array variable can coexist alongside a simple numerical variable of the same name because the array is always distinguished by its subscript).

The array elements' subscripts may be represented by any numerical expression yielding a valid subscript number. This means that an array can be processed using a FOR...NEXT loop. Thus, instead of the above long-winded program, we can now set up the variables $m(1)...m(10)$ using...

```
10 DIM m(10)
20 FOR n=1 TO 10
30 READ m(n)
40 NEXT n
50 DATA 75,44,90,38,55,64,70,12,75,60
```

Note that the DIM statement must come *before* any attempt to access the array in a program.

If you wish, you may examine the contents of the array by typing...

```
PRINT m(1)
PRINT m(2)
PRINT m(3)
etc...
```

You can also set up arrays with more than one dimension. In a two dimensional array you need two numbers to specify one of the elements - rather like the line and column numbers that specify a character position on the screen. If you imagine the line and column numbers (two dimensions) as referring to a printed page, you could then have an extra dimension for the page numbers. Of course, we are talking about numeric arrays; so the elements would not be printed characters as in a book, but numbers. Think of the elements of a three dimensional array v as being specified by $v(\text{page number}, \text{line number}, \text{column number})$.

For example, to set up a two-dimensional array c with dimensions 3 and 6, you use the DIM statement...

```
DIM c(3,6)
```

This then gives you $3*6=18$ subscripted variables...

```
c(1,1), c(1,2)... c(1,6)
c(2,1), c(2,2)... c(2,6)
c(3,1), c(3,2)... c(3,6)
```

The same principle works for any number of dimensions.

Although you can have a number and an array with the same name, you cannot have two arrays with the same name, even if they have a different number of dimensions.

There are also *string arrays*. The strings in an array differ from simple strings in that they are of *fixed length* and assignment to them is always Procrustean (ie. chopped off or padded with spaces).

The name of a string array is a single letter followed by \$. Unlike numeric arrays, a string array and a simple string variable *cannot* have the same name.

Suppose then, that you want an array a\$ of five strings. You must decide how long these strings are to be - let us suppose that 10 characters for each element is long enough. You then say...

```
DIM a$(5,10) (type this in)
```

This sets up a 5*10 array of characters, but you can also think of each row as being a string...

```
a$(1)=a$(1,1)a$(1,2)...a$(1,10)
a$(2)=a$(2,1)a$(2,2)...a$(2,10)
a$(3)=a$(3,1)a$(3,2)...a$(3,10)
a$(4)=a$(4,1)a$(4,2)...a$(4,10)
a$(5)=a$(5,1)a$(5,2)...a$(5,10)
```

If you give the same number of subscripts (two in this case) as there were dimensions in the DIM statement, then you get a single character; but if you miss the last one out, then you get a fixed length string. So, for instance, a\$(2,7) is the 7th character in the string a\$(2). Using the slicing notation, we could also write this as a\$(2)(7). Now type...

```
LET a$(2)="1234567890"
```

...and...

```
PRINT a$(2), a$(2,7)
```

You get...

```
1234567890      7
```

For the last subscript (the one you can miss out), you can also have a slicer, so that for instance...

```
a$(2,4 TO 8) is equal to a$(2)(4 TO 8) is equal to "45678"
```

Remember - In a string array, all the strings have the same, fixed length.

The DIM statement has an extra number (the last one) to specify this length. When you write down a subscripted variable for a string array, you can put in an extra number, or a *slicer*, to correspond with the extra number in the DIM statement.

You can have string arrays with no extra dimensions. Type...

```
DIM a$(10)
```

and you will find that a\$ behaves just like a string variable, except that it always has length 10, and assignment to it is always Procrustean.

Exercise...

1. Use READ and DATA statements to set up an array m\$ of twelve strings in which m\$(n) is the name of the nth month. (Hint - The DIM statement will be DIM m\$(12,9). Test it by printing out all the values of m\$(n) (use a loop)).

Part 13

Conditions

Subjects covered...

AND, OR
NOT

We saw in part 3 of this chapter how an I F statement takes the form...

I F condition THEN...

The conditions there were the relations (=, <, >, <=, >= and <>) which compare two numbers or two strings. You can also combine several of these, using the logical operations: AND, OR and NOT.

One relation AND another relation is true whenever *both* relations are true, so you could have a line like...

```
I F a$="yes" AND x>0 THEN PRINT x
```

...in which x only gets printed if a\$ is equal to 'yes' and x is greater than zero. The BASIC here is so close to English that it hardly seems worth spelling out the details. As in English, you can join lots of relations together with AND, and then the whole lot is true if all the individual relations are.

One relation OR another is true whenever *at least one* of the two relations is true. (Remember that it is still true if *both* the relations are true - this is something that English doesn't always imply.)

The NOT relationship turns things upside down. The NOT relation is true whenever the relation is false, and false whenever it is true.

Logical expressions may use combinations of AND, OR and NOT, just as numerical expressions may use combinations of +, -, * and so on. You can even put them in brackets if necessary. Logical operations have priorities in the same way as +, -, *, / and ↑ do. OR has the lowest priority, then AND, then NOT.

NOT is really a function, with an argument and a result, but its priority is much lower than that of other functions. Therefore, its argument does not need brackets unless it contains AND or OR (or both). NOT a = b means the same as NOT (a = b) (and the same as a <> b of course).

<> is the negation of = in the sense that it is true if, and only if, = is false. In other words...

a <> b is the same as NOT a = b

...and also...

NOT a <> b is the same as a = b

Convince yourself that >= and <= are the negations of < and > respectively. Thus you can always get rid of NOT from in front of a relation by changing the relation.

Also...

NOT (a first logical expression AND a second)

...is the same as...

NOT (the first) OR NOT (the second)

...and...

NOT (a first logical expression OR a second)

...is the same as...

NOT (the first) AND NOT (the second)

Using this, you can work NOTs through brackets until eventually they are all applied to relations, and then you can get rid of them. Logically speaking, NOT is unnecessary, although you might still find that using it makes a program clearer.

The following section is quite complicated, and can be skipped by the faint-hearted!

Try...

```
PRINT 1=2, 1<>2
```

...which you might expect to give a syntax error. In fact, as far as the computer is concerned, there is no such thing as a logical value - instead it uses ordinary numbers, subject to a few rules:

(i) =, <, >, <=, >= and <> all give the numeric results: 1 for true, and 0 for false. Thus, the PRINT command above printed 0 for '1=2', which is false, and 1 for '1<>2', which is true.

(ii) In the statement...

IF condition THEN...

...the condition can be actually any numeric expression. If its value is 0, then it counts as false, and any other value (including the value of 1 that a true relation gives) counts as true. Thus the IF statement means exactly the same as...

IF condition <> 0 THEN...

(iii) AND, OR and NOT are also number-valued operations...

x AND y has the value $\begin{cases} x, & \text{if } y \text{ is true (non-zero)} \\ 0 \text{ (false),} & \text{if } y \text{ is false (zero)} \end{cases}$

x OR y has the value $\begin{cases} 1 \text{ (true),} & \text{if } y \text{ is true (non-zero)} \\ x, & \text{if } y \text{ is false (zero)} \end{cases}$

NOT x has the value $\begin{cases} 0 \text{ (false),} & \text{if } x \text{ is true (non-zero)} \\ 1 \text{ (true),} & \text{if } x \text{ is false (zero)} \end{cases}$

(Notice that 'true' means 'non-zero' when we're checking a given value, but it means '1' when we're producing a new one).

Now try this program...

```
10 INPUT a
20 INPUT b
30 PRINT (a AND a>=b)+(b AND a< b)
40 GO TO 10
```

Each time it prints the larger of the two numbers a and b.

Convince yourself that you can think of...

x AND y

...as meaning...

x if y (else the result is 0)

...and of...

x OR y

...as meaning...

x unless y (in which case the result is 1)

An expression using AND or OR like this is called a conditional expression. An example using OR could be...

```
LET total=price less tax*(1.15 OR v$="zero rated")
```

Notice how AND tends to go with addition (because its default value is 0), and OR tends to go with multiplication (because its default value is 1).

You can also make string valued conditional expressions, but only using AND.

x\$ AND y has the value $\begin{cases} x\$ & \text{if } y \text{ is non-zero} \\ "" & \text{if } y \text{ is zero} \end{cases}$

...so it means x\$ if y (else the empty string).

Try this program, which inputs two strings and puts them in alphabetical order.

```
10 INPUT "type in two strings" 'a$,b$
20 IF a$>b$ THEN LET c$=a$: LET a$=b$: LET b$=c$
30 PRINT a$;" ";("<" AND a$<b$)+("=" AND a$=b$);" ";b$
40 GO TO 10
```

Part 14

The Character Set

Subjects covered...

CODE, CHR\$
POKE, PEEK
USR
BIN

The letters, digits, spaces, punctuation marks and so on that can appear in strings are called characters, and they make up the *character set* that the **+2** uses. Most of these characters are single symbols, but there are some more, called tokens, that represent whole words, such as PRINT, STOP, <> and so on.

There are 256 characters, and each one has a code between 0 and 255 (there is a complete list of them in part 27 of this chapter). To convert between codes and characters, there are two functions, CODE and CHR\$.

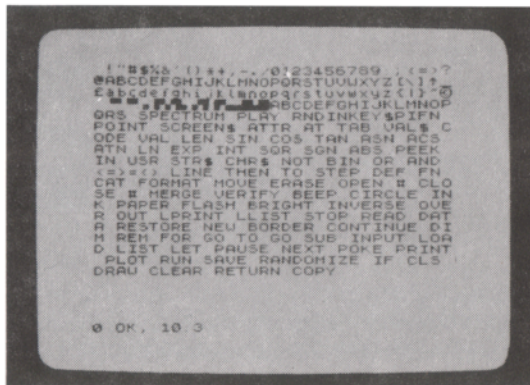
CODE is applied to a string, and gives the code of the first character in the string (or 0 if the string is empty).

CHR\$ is applied to a number, and gives the single character string whose code is that number.

This program prints out the entire character set...

```
10 FOR a=32 TO 255: PRINT CHR$ a;: NEXT a
```

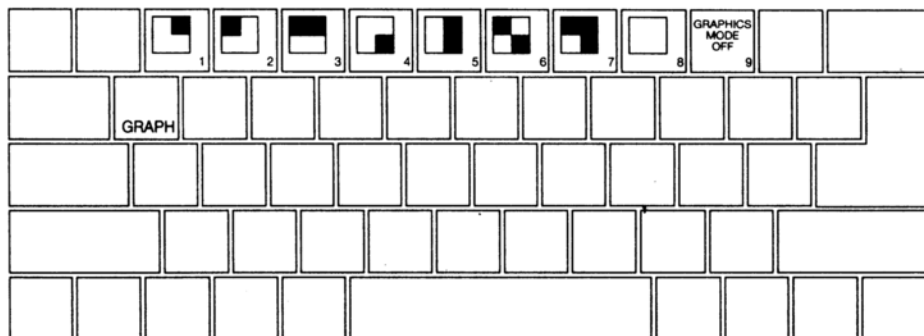
On the screen will appear the following...



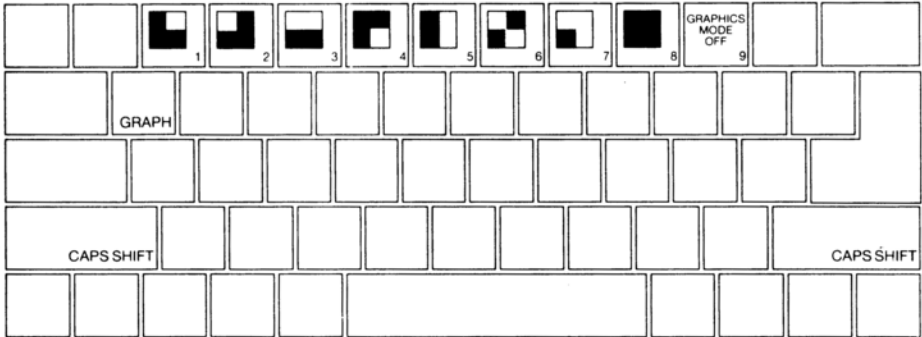
The character set

As you can see, the character set consists of a space, 15 symbols and punctuation marks, the ten digits, seven more symbols, the capital letters, six more symbols, the lower case letters and five more symbols. These are all (except £ and ©) taken from a widely-used set of characters known as ASCII (American Standard Codes for Information Interchange). ASCII also assigns numeric codes to these characters, and these are the codes that the +2 uses.

The rest of the characters are not part of ASCII, but are dedicated to the ZX Spectrum range of computers. First amongst them are a space and 15 patterns of black and white blobs. These are called the *graphics symbols* and can be used for drawing pictures. You can enter these from the keyboard, using what's known as *graphics mode*. Pressing the [GRAPH] key switches on graphics mode, after which the keys 1, 2, 3, 4, 5, 6, 7 and 8 will produce the graphics symbols...



















While in graphics mode, pressing **[CAPS SHIFT]** together with one of the keys **1** to **8** produces 'inverted' versions of the same symbols, ie. black becomes white and white becomes black...



The cursor keys won't work properly while all this is going on as the **+2** interprets them as shifted number keys, and prints graphics characters accordingly.

Pressing the **9** key turns everything back to normal (as does pressing **[GRAPH]** again). The **0** key deletes the character to the left of the cursor.

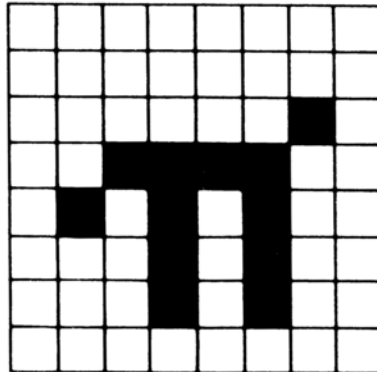
Here are the sixteen graphics symbols...

Symbol	Code	Symbol	Code
	128		143
	129		142
	130		141
	131		140
	132		139
	133		138
	134		137
	135		136

After the graphics symbols in the character set, you will see what appears to be another copy of the alphabet from A to S. These are characters that you can redefine yourself (though when the machine is first switched on they are set as letters) - they are called *user-defined graphics*. You can type these in from the keyboard by going into graphics mode, and then using the letter keys A to S.

To define a new character for yourself, follow this recipe - it defines a character to show π .

(i) Work out what the character looks like. Each character has an 8 x 8 grid of dots, each of which can appear to be either on or off. You'd draw a diagram something like this (with black squares representing the dots which are on)...



When a dot is on, the `+2` prints the ink colour; when a dot is off, the `+2` prints the paper colour. (The terms *ink* and *paper* are explained in part 16 of this chapter.)

We've left a one-square border around the edge of the character because all the other letters also have one (except for lower case letters with tails, where the tail goes right down to the bottom).

(ii) Work out which user-defined graphic you wish to display π - let's say the one corresponding to **P**, so that if you press **P** (after pressing **[GRAPH]**) you get π .

(iii) Store the new pattern. Each user-defined graphic has its pattern stored as eight numbers, one for each row. You can write each of these numbers in a program as **B I N** followed by eight 0's or 1's - 0 for paper, 1 for ink - so the eight numbers for our π character are...

```

B I N 00000000 - top row
B I N 00000000 - second row down
B I N 00000010 - third row down
B I N 00111100 - fourth row down
B I N 01010100 - fifth row down
B I N 00010100 - sixth row down
B I N 00010100 - seventh row down
B I N 00000000 - bottom row

```

(If you know about binary numbers, then it should help you to know that **B I N** is used to write a number in binary instead of the usual decimal.) Look at the pattern of binary numbers through half-closed eyes - you may even be able to see the π character.

These eight numbers are stored in eight locations (bytes) in memory. Each of these locations has an *address*. The address of the first byte (or group of eight digits) is `USR "P"` (we chose *P* in (ii) above). The address of the second byte is `USR "P" + 1`, and so on up to the eighth byte, which has the address `USR "P" + 7`.

USR here is a function to convert a string argument into the address of the first byte in memory for the corresponding user-defined graphic. The string argument must be a single character which can be either the user-defined graphic itself or the corresponding letter (in upper or lower case). There is another use for USR, when its argument is a number, which will be dealt with later.

Even if you don't understand this, the following program will define the character for you...

```
10 FOR n=0 to 7
20 READ row: POKE USR "P"+n,row
30 NEXT n
40 DATA BIN 00000000
50 DATA BIN 00000000
60 DATA BIN 00000010
70 DATA BIN 00111100
80 DATA BIN 01010100
90 DATA BIN 00010100
100 DATA BIN 00010100
110 DATA BIN 00000000
```

The POKE statement stores a number directly in a memory location, bypassing the mechanisms normally used by the BASIC. The opposite of POKE is PEEK, and this allows us to look at the contents of a memory location although it does not actually alter the contents themselves. PEEK and POKE are described more fully in part 24 of this chapter.

After the user-defined graphics in the character set come the *tokens*.

You will have noticed that we have not printed out the first 32 characters (codes 0 to 31) - these are *control characters*. They don't produce anything printable, but instead are used to control the screen display or some other function of the +2.

(If you try to print control characters, the +2 displays ? to show that it doesn't understand them. Control characters are described more fully in part 27 of this chapter.)

The three codes that the screen display uses are 6, 8 and 13 (these will now be explained). On the whole, CHR\$ 8 is the only one you are likely to find useful.

CHR\$ 6 prints spaces in exactly the same way as a comma does in a PRINT statement, for instance...

```
PRINT 1; CHR$ 6;2
```

...does the same as...

```
PRINT 1,2
```

Obviously this is not a very clear way of using it. A more subtle way is to say...

```
LET a$="1"+CHR$ 6+"2"
PRINT a$
```

CHR\$ 8 is 'backspace' - it moves the print position back one place - try...

```
PRINT "1234"; CHR$ 8;"5"
```

...which prints out...

```
1235
```

CHR\$ 13 is 'newline' - it moves the print position to the beginning of the next line.

The screen display also uses control codes 16 to 23 - these are explained in parts 15 and 16 of this chapter (all the codes are listed in part 27).

Using the codes for the characters we can extend the concept of 'alphanumerical ordering' to cover strings containing any characters, not just letters. If instead of thinking in terms of the usual alphabet of 26 letters we use the extended alphabet of 256 characters, in the same order as their codes, then the principle is exactly the same. For instance, the following strings are in their 'Spectrum' alphabetical order. (Notice the rather odd feature that lower case letters come after all the capitals; so 'a' comes after 'Z'; also, spaces are significant.)

```
CHR$ 3+"ZOOLOGICAL GARDENS"  
CHR$ 8+"AARDVARK HUNTING"  
" AAAARGH!"  
"(Parenthetical remark)"  
"100"  
"129.95 inc. VAT"  
"AASVOGEL"  
"Aardvark"  
"Elgar, the Regal Lager"  
"PRINT"  
"Zoo"  
"[interpolation]"  
"aardvark"  
"aasvogel"  
"derby"  
"zoo"  
"zoology"
```

Here is the rule for finding out which order two strings come in. First, compare the first characters. If they are different, then one of them has its code less than the other, and the string it came from is the earlier (lesser) of the two strings. If they are the same, then go on to compare the next characters. If in this process one of the strings runs out before the other, then that string is the earlier; otherwise they must be equal.

The relations =, <, >, <=, >=, and <> are used for strings as well as for numbers: < means 'comes before' and > means 'comes after', so that...

```
"AA man"<"AARDVARK"  
"AARDVARK">"AA man"
```

...are both true.

<= and >= work the same way as they do for numbers, so that...

"The same string"<="The same string"

...is true, but...

"The same string"<"The same string"

...is false.

Experiment on all this using the program here, which inputs two strings and puts them in order.

```
10 INPUT "Type in two strings:",a$,b$
20 IF a$>b$ THEN LET c$=a$: LET a$=b$: LET b$=c$
30 PRINT a$;" ";
40 IF a$<b$ THEN PRINT "<";: GO TO 60
50 PRINT "=";
60 PRINT " ";b$
70 GO TO 10
```

Note (in the above program and in the program at the end of part 13) how we have to introduce c\$ in line 20 when we swap over a\$ and b\$. Can you see why simply using...

```
LET a$=b$: LET b$=a$
```

...would not have the desired effect?

The next program sets up user defined graphics for the following keys to display chess pieces...

B for bishop
K for king
R for rook
Q for queen
P for pawn
N for knight

Chess pieces...

```
5 LET b=BIN 01111100: LET c=BIN 00111000:
LET d=BIN 00010000
10 FOR n=1 TO 6: READ p$: REM 6 pieces
20 FOR f=0 TO 7: REM read pieces into 8 bytes
30 READ a: POKE USR p$+f,a
40 NEXT f
50 NEXT n
100 REM bishop
110 DATA "b",0,d, BIN 00101000, BIN 01000100
120 DATA BIN 01101100,c,b,0
130 REM king
140 DATA "k",0,d,c,d
```

```

150 DATA c, BIN 01000100,c,0
160 REM rook
170 DATA "r",0, BIN 01010100,b,c
180 DATA c,b,b,0
190 REM queen
200 DATA "q",0, BIN 01010100, BIN 00101000,d
210 DATA BIN 01101100,b,b,0
220 REM pawn
230 DATA "p",0,0,d,c
240 DATA c,d,b,0
250 REM knight
260 DATA "n",0,d,c, BIN 01111000
270 DATA BIN 00011000,c,b,0

```

Note that in the above DATA statements, we have simply used 0 instead of BIN 00000000.

When you have run this program, you may look at the pieces by pressing [GRAPH] followed by any of the keys: B, K, R, Q, P or N.

Exercises...

1. Imagine the space for one symbol divided up into four quarters like a Battenburg cake. Then if each quarter can be either black or white, there are $2^4 = 16$ possibilities. Find them all in the character set.
2. Run this program...

```

10 INPUT a
20 PRINT CHR$ a;
30 GO TO 10

```

If you experiment with it, you'll find that CHR\$ a is rounded to the nearest whole number; and if a is not in the range 0 to 255, then the program stops with the error report B i n t e g e r o u t o f r a n g e.

3. Which of these is the lesser?

```

"EVIL"
"evil"

```

Part 15

More about PRINT and INPUT

Subjects covered...

C L S

PRINT items: nothing at all

Expressions (numeric or string type): T A B numeric expressions, A T numeric expression

PRINT separators: , ; '

INPUT items: variables (numeric or string type)

L I N E string variable

Any PRINT item not beginning with a letter. (Tokens are not considered as beginning with a letter.)

Scrolling

S C R E E N \$

You have already seen PRINT used quite a lot, so you will have a rough idea of how it is used. Expressions whose values are printed are called PRINT *items*. They may be separated by commas, semicolons or apostrophes, which are called PRINT *separators*. A PRINT item can also be nothing at all, which is a way of explaining what happens when you use two commas in a row.

there are two more kinds of PRINT items, which are used to tell the +2 not *what*, but *where* to print. For example, the instruction...

```
10 PRINT AT 11,16;"*"
```

...prints a star in the centre of the screen. This is because...

A T line , column

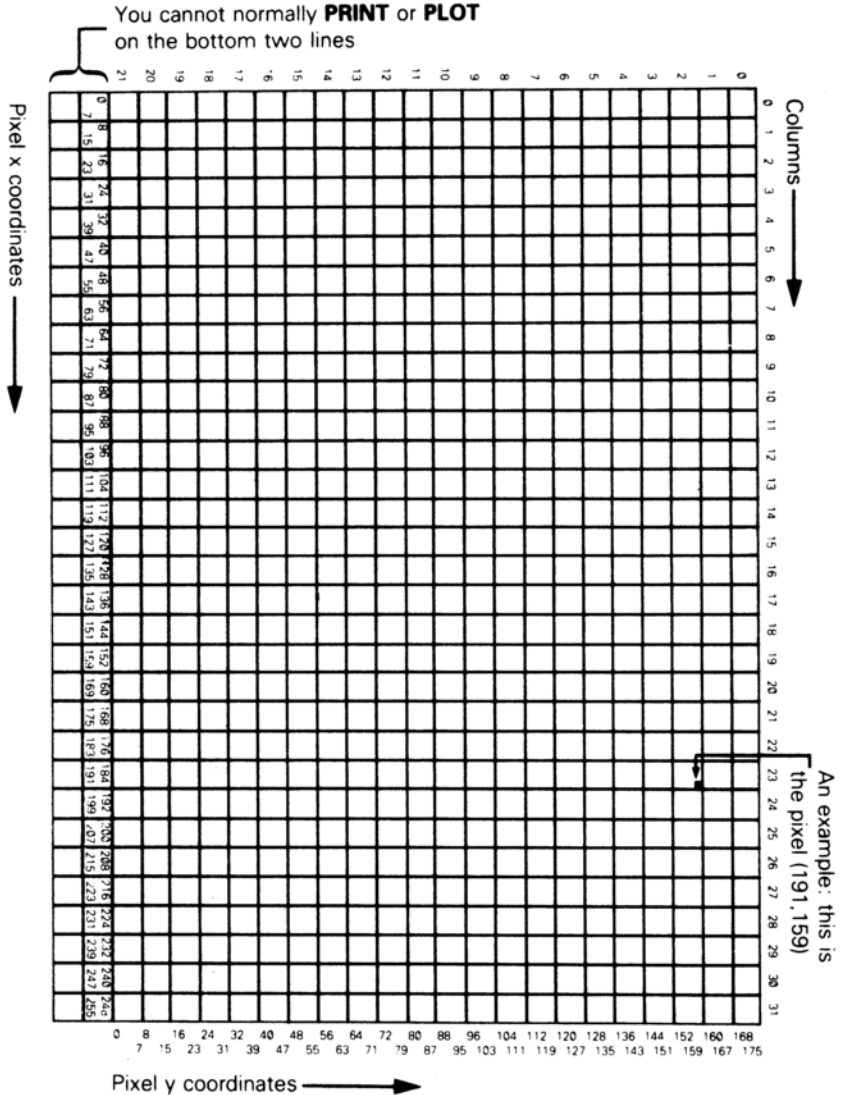
...moves the PRINT position (the place where the next item is to be printed) to the line and column specified. Lines are numbered from 0 (at the top) to 21; columns are numbered from 0 (on the left) to 31.

S C R E E N \$ is the reverse function to PRINT AT, and will (within limits) 'read' the character which is located at a particular position on the screen. It uses line and column numbers in the same way as PRINT AT, but enclosed in brackets. For example, the instruction...

```
20 PRINT AT 0,0; SCREEN$ (11,16)
```

...will read the star printed in the centre of the screen, then print it at location 0,0 (the top left hand corner).

Characters from tokens are read normally (as single characters), and spaces are read as spaces. Attempting to read user-defined characters, graphics characters, or lines drawn by PLOT, DRAW and CIRCLE, however, result in a null (empty) string being returned. The same applies if OVER has been used to create a composite character. (The keywords PLOT, DRAW, CIRCLE and OVER are described in parts 16 and 17 of this chapter.)



The function...

TAB column

...prints enough spaces to move the PRINT position to the column specified. It stays on the same line, or, if this would involve backspacing, moves to the next line. Note that the +2 reduces the column number 'modulo 32' (ie. it divides by 32 and takes the remainder) - so TAB 33 means the same as TAB 1.

As an example...

```
PRINT TAB 30;1; TAB 12;"Contents"; AT 3,1;"Chapter";  
TAB 24;"Page"
```

...is how you might want to print out the heading on the contents page (page 1) of a book.

Try running this...

```
10 FOR n=0 TO 20  
20 PRINT TAB 8*n;n;  
30 NEXT n
```

This shows what is meant by the TAB numbers being reduced modulo 32.

For a more elegant example, change the 8 in line 20 to a 6.

Note the following points:

- (i) TABs and print items are best terminated with semicolons, as we have done above. You can use commas (or nothing, at the end of the statement), but this means that after having carefully set up the PRINT position, you immediately move it on again - not terribly useful!
- (ii) You cannot print on the bottom two lines (22 and 23) on the screen because they are reserved for commands, INPUT data, error messages, reports and so on. References to 'the bottom line' usually mean line 21.
- (iii) You can use AT to locate the PRINT position even where there is already something printed - the new print item will simply overwrite the old.

Another statement connected with PRINT is CLS. This clears the whole screen.

When printing reaches the bottom of the screen, it starts to scroll upwards rather like a typewriter. You can see this if you go into the small screen using the edit menu option 'Screen' (described in chapter 6), and then type...

```
CLS: FOR n=1 TO 30: PRINT n: NEXT n
```

When it has printed a screen full, the +2 will stop with the message scroll? at the bottom of the screen. You can now inspect the first 22 numbers at your leisure. When you have finished with them, press Y (for 'yes') and the +2 will give you the next screen full of numbers. Actually, any key will

make the +2 carry on except N (for 'no'), the [BREAK] key or the space bar. These will make the +2 stop running the program with the report D B R E A K - C O N T r e p e a t s .

The INPUT statement can do much more than we have told you so far. You have already seen INPUT statements like...

```
INPUT "How old are you?", age
```

...in which the +2 prints the caption 'How old are you?' at the bottom of the screen, and then you have to type in your age. In fact though, an INPUT statement can be made up of items and separators in exactly the same way as a PRINT statement, so 'How old are you?' and 'age' are both INPUT items. INPUT items are generally the same as PRINT items, however, there are some very important differences:

First, an obvious extra INPUT item is the *variable* whose value you require to be typed in - age in our example above. The rule is that if an INPUT item begins with a letter, then it must be a variable whose value is to be input.

This would seem to mean that you can't print out the values of variables as part of a caption. However, you can get round this by putting brackets around the variable. Any expression that starts with a letter must be enclosed in brackets if it is to be printed as part of a caption.

Any kind of PRINT item that is not affected by these rules is also an INPUT item. Here is an example to illustrate what's going on...

```
LET my age = INT (RND * 100): INPUT ("I am ";my age;".");  
"How old are you?", your age
```

my age is contained in brackets, so its value gets printed out. your age is *not* contained in brackets, so you have to type its value in.

Everything that an INPUT statement writes goes to the bottom part of the screen, which acts somewhat independently of the top half. In particular, its lines are numbered relative to the top line of the bottom half, even if this has moved up the actual TV screen (which it does if you type lots of INPUT data). Whatever the small screen does during INPUT, however, it will always revert to being two lines in size when the program stops, and you start editing.

To see how AT works in INPUT statements, try this...

```
10 INPUT "This is line 1.",a$; AT 0,0;"This is line 0.",a$;  
AT 2,0;"This is line 2.",a$; AT 1,0;"This is still line  
1.",a$
```

Run the program (just press [ENTER] each time it stops). When 'This is line 2', is printed, the lower part of the screen moves up to make room for it; but the numbering moves up as well, so that the lines of text keep their same numbers.

Now try this...

```
10 FOR n=0 TO 19: PRINT AT n,0;n:: NEXT n
20 INPUT AT 0,0;a$: AT 1,0;a$: AT 2,0;a$: AT 3,0;a$:
   AT 4,0;a$: AT 5,0;a$;
```

As the lower part of the screen goes up and up, the upper part remains undisturbed until the lower part threatens to write on the same line as the PRINT position. Then the upper part starts scrolling up to avoid this.

Another refinement to the INPUT statement that we haven't seen yet is called LINE input and is a different way of inputting string variables. If you use LINE before the name of a string variable to be input, as in...

```
INPUT LINE a$
```

...then the +2 will not give you the string quotes that it normally does for a string variable (though it will pretend to itself that they are there). So if you type in...

```
cat
```

...as the INPUT data, a\$ will be given the value 'cat'. Because the string quotes do not appear with the string, you cannot delete them and type in a different sort of string expression for the INPUT data. Remember that you cannot use LINE for numeric variables.

There's an interesting side effect to INPUT. Whilst typing into an INPUT request, the old Spectrum single-key entry system enjoys a brief moment of freedom before being locked away again when you press [ENTER]. Run this program if you're interested...

```
10 INPUT numbers
20 PRINT numbers
30 GO TO 10
```

Input a few numbers, and they'll get printed faithfully onto the screen. Now press [EXTEND MODE] followed by the M key. The word PI appears, and if you press [ENTER], then 3.1415927 will appear as if by magic. However, if you type PI as two letters without the aid of [EXTEND MODE] then the +2 will stop with the report...

```
2 Variable not found, 10:1
```

There's no simple explanation for this behaviour, and it's best just to be aware that it can happen if you press some combinations of keys during INPUT. If for some reason you're keen to experiment, chapter 7 will tell you which keys produce which effects.

The control characters CHR\$ 22 and CHR\$ 23 have effects rather like AT and TAB. Whenever the +2 is instructed print one of them, the character must be followed by two more characters that do not have their usual effect, but that are treated instead as numbers (their codes) to specify the line and column (for AT) or the tab position (for TAB). You will almost always find it easier to use AT and TAB in the usual way rather than use control characters, however, they might be useful in some

circumstances. The AT control character is CHR\$ 22. The first character after it specifies the line number and the second specifies the column number, so that...

```
PRINT CHR$ 22+CHR$ 1+CHR$ c;
```

...has exactly the same effect as...

```
PRINT AT 1,c;
```

This is so that even if CHR\$ 1 or CHR\$ c would normally have a different meaning (for instance if c=13); the CHR\$ 22 before them overrides that.

The TAB control character is CHR\$ 23 and the two characters after it combine to give a number between 0 and 65535, specifying the number you would have in a TAB item. The statement...

```
PRINT CHR$ 23+CHR$ a+CHR$ b;
```

...has the same effect as...

```
PRINT TAB a+256*b;
```

You can use POKE to stop the computer asking if you wish to scroll? by typing...

```
POKE 23692,255
```

...every so often. After this it will scroll up 255 times before stopping with scroll? As an example, try...

```
10 FOR n=0 TO 1000
20 PRINT n: POKE 23692,255
30 NEXT n
```

...and watch everything whizz off the screen!

Exercise...

1. Try this program on some children, to test their multiplication tables...

```
10 LET m$=""
20 LET a=INT (RND*12)+1: LET b=INT (RND*12)+1
30 INPUT (m$) ' ' "what is ";(a);" x ";(b);"?:";c
100 IF c=a*b THEN LET m$="Right.": GO TO 20
110 LET m$="Wrong. Try again.": GO TO 30
```

If they are perceptive, they might manage to work out that they do not have to do the calculation themselves. For instance, if the +2 asks them to type the answer to 2 x 3, then all they have to do is type in 2 * 3 literally.

Part 16

Colours

Subjects covered...

INK, PAPER, FLASH, BRIGHT, INVERSE, OVER
BORDER

Run this program...

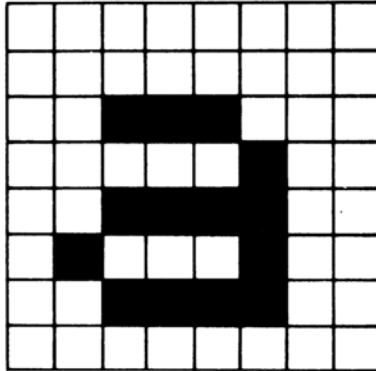
```
10 FOR m=0 TO 1: BRIGHT m
20 FOR n=1 TO 10
30 FOR c=0 TO 7
40 PAPER c: PRINT " ";: REM 4 coloured spaces
50 NEXT c: NEXT n: NEXT m
60 FOR m=0 TO 1: BRIGHT m: PAPER 7
70 FOR c=0 TO 3
80 INK c: PRINT c;" ";
90 NEXT c: PAPER 0
100 FOR c=4 TO 7
110 INK c: PRINT c;" ";
120 NEXT c: NEXT m
130 PAPER 7: INK 0: BRIGHT 0
```

This shows the eight colours (including white and black) and the two levels of brightness that the **+2** can produce on a colour television. (If your TV is black-and-white, then you will just see various shades of grey.) A quicker way to achieve a similar result is to **RESET** the **+2** whilst holding down **[BREAK]** - but that's a little drastic. Here is a list of which numbers produce which colours (for your reference)...

- 0 - black
- 1 - blue
- 2 - red
- 3 - magenta
- 4 - green
- 5 - cyan
- 6 - yellow
- 7 - white

On a black-and-white TV, these numbers are in order of brightness. To use these colours properly, you need to understand a bit about how the picture is arranged.

The picture is divided up into 768 (24 lines of 32) positions (*cells*) where characters can be printed.



A typical character cell

Each character cell consists of an 8 x 8 grid (such as above). This should remind you of the user-defined graphics in part 14, where we had 0s for the white dots and 1s for the black dots.

The character has two colours associated with it: the *ink*, or foreground colour, which is the colour for the black dots in our square, and the *paper*, or background colour, which is used for the white dots. To start off with, every cell has black ink and white paper so writing appears as black on white.

The character also has a brightness (normal or extra bright), and something to say whether it flashes or not. Flashing is done by continuously swapping the ink and paper colours. All this information can be coded into numbers, so a character then has the following...

- (i) An 8 x 8 grid of 0s and 1s to define the shape of the character, with 0 for paper and 1 for ink.
- (ii) Ink and paper colours, each coded into a number between 0 and 7.
- (iii) A brightness - 0 for normal, 1 for extra bright.
- (iv) A flash number - 0 for steady, 1 for flashing.

Note that since the ink and paper colours cover a whole character cell, you cannot possibly have more than two colours in a given block of 64 dots. The same goes for the brightness and flash numbers - they refer to the whole character cell, not individual dots within the cell. The colour, brightness and flash number for a given character cell are called *attributes*.

When you print something on the screen, you change the dot pattern for that character cell. It is less obvious, but still true, that you also change the cell's attributes. To start off with you do not notice this because everything is printed with black ink on white paper (at normal brightness and no flashing); however, you can vary this with the INK, PAPER, BRIGHT and FLASH statements. Using the edit menu's 'S c r e e n' option, go to the bottom screen, and try...

PAPER 5

...and then PRINT a few items on the screen - they will appear on cyan paper, because as they are printed, the paper colour for the cells they occupy are set to cyan (which has code 5).

The others work the same way, so you may use the settings...

PAPER	(whole number between 0 and 7)
INK	(whole number between 0 and 7)
BRIGHT	(whole number between 0 and 1)
FLASH	(whole number between 0 and 1)

...and any printing will set the corresponding attributes for all the character cells it subsequently uses.

Try some of these out. You should now be able to see how the program at the beginning of this section worked (remember that a space is a character that has its ink and paper the same colour).

There are some more numbers you can use in these statements that have less direct effects.

8 can be used in all four statements, and means 'transparent' in the same sense that the old attribute shows through. Suppose, for instance, that you do...

PAPER 8

No character position will ever have its paper colour set to 8 because there is no such colour; what happens is that when a position is printed on, its paper colour is left the same as it was before. However, INK 8, BRIGHT 8 and FLASH 8 work the same way as for the other attribute numbers.

9 can be used only with PAPER and INK, and means 'contrast'. The colour (ink or paper) that you use it with is made to contrast with the other by being made white if the other is a dark colour (black, blue, red or magenta), or being made black if the other is a light colour (green, cyan, yellow or white).

Try this by doing...

```
INK 9: FOR c=0 TO 7: PAPER c: PRINT c: NEXT c
```

A more impressive display of its power is to run the program at the beginning to make coloured stripes (again, making sure that you are in the lower screen when you type RUN), and then doing...

```
INK 9: PAPER 8: PRINT AT 0,0; FOR n=1 TO 1000: PRINT n; :  
NEXT n
```

The ink colour here is always made to contrast with the old paper colour for each character cell.

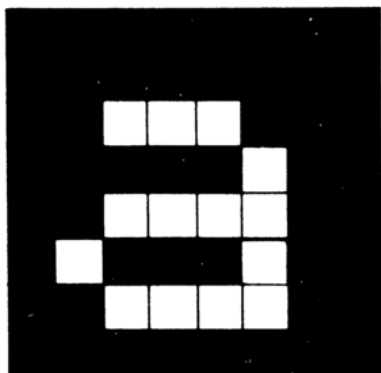
Colour TV relies on the rather curious fact that the human eye can only really see three colours - red, green and blue. The other colours are mixtures of these. For instance, magenta is made by mixing red with blue - which is why its code, 3, is the sum of the codes for red and blue.

To see how all eight colours fit together, imagine three rectangular spotlights, coloured red, green and blue shining at not quite the same place on a piece of white paper in the dark. Where they overlap you will see mixtures of colours, as shown by the following program (note that solid ink spaces are obtained by entering graphics mode (pressing [GRAPH]) then holding down [CAPS SHIFT] while pressing 8. To exit from graphics mode, press 9.)

```
10 BORDER 0: PAPER 0: INK 7: CLS
20 FOR a=1 TO 6
30 PRINT TAB 6; INK 1; " ██████████ ": REM 18
   ink squares
40 NEXT a
50 LET dataline=200
60 GO SUB 1000
70 LET dataline=210
80 GO SUB 1000
90 STOP
200 DATA 2,3,7,5,4
210 DATA 2,2,6,4,4
1000 FOR a=1 TO 6
1010 RESTORE dataline
1020 FOR b=1 TO 5
1030 READ c: PRINT INK c; " ██████████ ";: REM 6
   ink squares
1040 NEXT b: PRINT: NEXT a
1050 RETURN
```

There is a function called ATTR that finds out what the attributes are at a given position on the screen. It is a fairly complicated function, so it has been relegated to the end of this section.

There are two more statements, INVERSE and OVER, which control not the attributes, but the dot pattern that is printed on the screen. They use the numbers 0 for off, and 1 for on. If you use INVERSE 1, then each character cell's dot pattern will be the inverse of its usual form, ie. paper dots will be replaced by ink dots and vice versa. Thus the character cell containing 'a' (shown previously) would be printed as follows (on the next page)...



If (as at switch on) we have black ink and white paper, then the 'a' will appear as white on black.

The statement...

```
OVER 1
```

...sets into action a particular sort of overprinting. Normally when something is written into a character position it completely obliterates what was there before; however, using `OVER 1`, the new character is simply added on top of the old one. This can be particularly useful for writing composite characters, like an underlined letter, as in the following program. (Reset the `+2` and select 128 BASIC. Note that the underline character is obtained by `[SYMB SHIFT]` with `0`.)

```
10 OVER 1
20 PRINT "w"; CHR$ 8;"_";
```

(Notice we have used the control character `CHR$ 8` (backspace) before overprinting the `w` with `_`.)

There is another way of using `INK`, `PAPER` and so on which you will probably find more useful than having them as statements. You can put them as items in a `PRINT` statement (followed by `;`), and they then do exactly the same as they would have done if they had been used as statements on their own, except that their effect is only temporary, lasting as far as the end of the `PRINT` statement that contains them. Thus if you type...

```
PRINT PAPER 6;"x";: PRINT "y"
```

...then only the `x` will be on yellow paper.

INK and the rest when used as statements do not affect the colour in the bottom part of the screen, where INPUT data is typed in and errors are displayed. The bottom screen uses the colour of the border for its paper colour, code 9 (for contrast) for its ink colour, has flashing off, and everything at normal brightness. You can change the border colour to any of the eight normal colours (not 8 or 9) using the statement...

BORDER colour

When you type in INPUT data, it follows this rule of using contrasting ink on border coloured paper; but you can change the colour of the captions written by the +2 by using INK and PAPER (and so on) items in the INPUT statement, just as you would in a PRINT statement. Their effect lasts either to the end of the statement, or until some INPUT data is typed in, whichever comes soonest. Try...

```
INPUT FLASH 1; INK 1;"What is your number?";n
```

The +2 has a high regard for your sanity - no matter what combination of effects and colours you manage to produce from a BASIC program, the editor will always use black ink on white paper.

There is one more way of changing the colours by using control characters - rather like the control characters for AT and TAB in part 15.

```
CHR$ 16 corresponds to INK
CHR$ 17 corresponds to PAPER
CHR$ 18 corresponds to FLASH
CHR$ 19 corresponds to BRIGHT
CHR$ 20 corresponds to INVERSE
CHR$ 21 corresponds to OVER
```

These are each followed by one character that shows a colour by its code; so that (for instance)...

```
PRINT CHR$ 16+CHR$ 9;"item"
```

...has the same effect as...

```
PRINT INK 9;"item"
```

On the whole, you would not bother to use these control characters because you might just as well use the statements INK, PAPER, etc. However, if you have some old 48K BASIC programs on cassette, you may find such control characters embedded in the listing. In general, the editor will actively ignore them, and remove them at the first opportunity. It is not possible to insert them into listings as with the old 48K Spectrum.

The ATTR function has the form...

```
ATTR (line, column)
```

Its two arguments are the line and column numbers that you would use in an AT item, and its result is a number that shows the colours and so on at the corresponding character position on the TV screen. You can use this as freely in expressions as you can any other function.

The number that is the result is the sum of four other numbers as follows:

128 if the character cell is flashing, 0 if it is steady.
64 if the character cell is bright, 0 if it is normal.
8 multiplied by the code for the paper colour.
1 multiplied by the code for the ink colour.

For instance, if the character cell is flashing, normal brightness, yellow paper and blue ink, then the four numbers that we have to add together are 128, 0, $8*6=48$ and 1, making 177 altogether. Test this with...

```
PRINT AT 0,0; FLASH 1; PAPER 6; INK 1;" ";ATTR (0,0)
```

Exercises...

1. Try...

```
PRINT "B"; CHR$ 8; OVER 1;"/";
```

Where the / has cut through the B, it has left a white dot. This is the way overprinting works on the +2 -two papers or two inks give a paper, one of each gives an ink. This has the interesting property that if you overprint with the same thing twice you get back what you started off with. If you now type...

```
PRINT CHR$ 8; OVER 1;"/"
```

...why do you recover an unblemished B?

2. Run this program...

```
10 POKE 22527+RND*704, RND*127  
20 GO TO 10
```

(Never mind how this program works) The program is changing the colours of squares on the TV screen and the RND should ensure that this happens randomly. The diagonal stripes that you eventually see are a manifestation of the hidden pattern in RND, ie. pseudo-random instead of truly random.

Part 17

Graphics

Subjects covered...

PLOT, DRAW, CIRCLE
pixels

For all of this section, type in the example programs, commands and RUN in the small screen (use the edit menu's 'S c r e e n' option).

In this section we shall see how to draw pictures on the +2. The part of the screen you can use has 22 lines and 32 columns, making $22 \times 32 = 704$ character positions. As you may remember from part 16, each of these character positions is made up of an 8 x 8 grid of dots which are called *pixels* (picture elements).

A pixel is specified by two numbers, - its coordinates. The first, its *x* coordinate, says how far it is across from the extreme left hand column. The second, its *y* coordinate, says how far it is up from the bottom. These coordinates are usually written as a pair in brackets, so (0,0), (225,0), (0,175) and (255,175) are the bottom left, bottom right, top left and top right corners of the screen.

If you have trouble memorising which coordinate is which, simply remember that *x is a cross* (*x* is across).

The statement...

```
PLOT x coordinate , y coordinate
```

...links in the pixel with these coordinates, so this measles program...

```
10 PLOT INT (RND*256) , INT (RND*176) : INPUT a$: GO TO 10
```

...plots a random point each time you press [ENTER].

Here is a rather more interesting program. It plots a graph of the function SIN (a sine wave) for values between 0 and 2π ...

```
10 FOR n=0 TO 255  
20 PLOT n,88+80*SIN (n/128*PI)  
30 NEXT n
```

This next program plots a graph of SQR (part of a parabola) between 0 and 4...

```
10 FOR n=0 TO 255  
20 PLOT n,80*SQR (n/64)  
30 NEXT n
```

Notice that pixel coordinates are rather different from the line and column in an AT item. You may find the diagram in part 15 of this chapter useful when working out pixel coordinates and line and column numbers.

To help you with your pictures, the **+2** will draw straight lines, circles and parts of circles for you, using the **DRAW** and **CIRCLE** statements.

The statement **DRAW** (to draw a straight line) takes the form...

```
DRAW x,y
```

The starting place of the line is the pixel where the last **PLOT**, **DRAW** or **CIRCLE** statement left off (this is called the **PLOT** position - **RUN**, **CLEAR**, **CLS** and **NEW** reset it to the bottom left hand corner, at 0,0); the finishing place of the line is *x* pixels to the right of that and *y* pixels up. The **DRAW** statement on its own determines the length and direction of the line, but not its starting point.

Experiment with a few **PLOT** and **DRAW** commands, for instance...

```
PLOT 0,100: DRAW 80,-35
PLOT 90,150: DRAW 80,-35
```

Notice that the numbers in a **DRAW** statement can be negative, but those in a **PLOT** statement can't.

You can also plot and draw in colour, although you have to bear in mind that colours always cover the whole of a character cell and cannot be specified for individual pixels. When a pixel is plotted, it is set to show the full ink colour, and the whole of the character cell containing it is given the current ink colour. This program demonstrates that point...

```
10 BORDER 0: PAPER 0: INK 7: CLS: REM black out screen
20 LET x1=0: LET y1=0: REM start of line
30 LET c=1: REM for ink colour, starting blue
40 LET x2=INT (RND*256): LET y2=INT (RND*176): REM random
   finish on line
50 DRAW INK c;x2-x1,y2-y1
60 LET x1=x2: LET y1=y2: REM next line starts where last one
   finished
70 LET c=c+1: IF c=8 THEN LET c=1: REM new colour
80 GO TO 40
```

The lines seem to get broader as the program goes on, and this is because a line changes the colours of all the inked-in pixels of all the character cells that it passes through. Note that you can embed **PAPER**, **INK**, **FLASH**, **BRIGHT**, **INVERSE** and **OVER** items in a **PLOT** or **DRAW** statement just as you could with **PRINT** and **INPUT**. They go between the keyword and the coordinates, and are terminated by either semicolons or commas.

An extra frill with **DRAW** is that you can use it to draw parts of circles instead of straight lines, by including an extra number to specify an angle to be turned through. The form is...

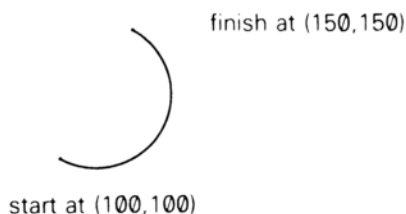
```
DRAW x,y,a
```

x and y are used to specify the finishing point of the line just as before, and a is the number of radians that it must turn through as it goes. If a is positive then it turns to the left; if a is negative then it turns to the right. Another way of seeing a is as showing the fraction of a complete circle that will be drawn, (a complete circle is 2π radians) so if $a=\pi$ it will draw a semicircle, if $a=0.5\pi$ a quarter of a circle, and so on.

For instance, suppose $a=\pi$. Then whatever values x and y take, a semicircle will be drawn. Try...

```
10 PLOT 100,100: DRAW 50,50,PI
```

...which will draw this...



The drawing starts off in a south-easterly direction, but by the time it stops, it is going north-west. In between, it has turned through 180 degrees, or π radians (the value of a).

Run the program several times, with `PI` replaced by various other expressions, eg. `-PI`, `PI/2`, `3*PI/2`, `PI/4`, `1`, `0`, etc.

The last statement in this section is `CIRCLE`, which draws an entire circle. You specify the coordinates of the centre and the radius of the circle using...

```
CIRCLE x coordinate , y coordinate , radius
```

Just as with `PLOT` and `DRAW`, you can put the various sorts of colour items in at the beginning of a `CIRCLE` statement.

The `POINT` function tells you whether a pixel is ink or paper colour. Its two arguments are the coordinates of the pixel (which must be enclosed in brackets) and its result is 0 if the pixel is paper colour; or 1 if it is ink colour. Try...

```
CLS : PRINT POINT (0,0): PLOT 0,0: PRINT POINT (0,0)
```

Type...

```
PAPER 7: INK 0
```

...and let us investigate how INVERSE and OVER work inside a PLOT statement. These two affect just the relevant pixel, and not the rest of the character cell. They are normally off (0) in a PLOT statement, so you only need to mention them to turn them on (1).

Here is a list of the possibilities for reference:

PLOT ; - This is the usual form. It plots an ink dot, ie. sets the pixel to show the ink colour.

PLOT INVERSE 1 ; - This plots a dot of 'ink eradicator', ie. it sets the pixel to show the paper colour.

PLOT OVER 1 ; - This exchanges the pixel colour with whatever it was before, so if it was ink colour then it becomes paper colour, and vice versa.

PLOT INVERSE 1 ; OVER 1 ; - This leaves the pixel exactly as it was before, but note that it also changes the PLOT position, so you might use it simply to do that.

As another example of using the OVER statement, fill the screen up with writing using black on white, and then type...

```
PLOT 0,0: DRAW OVER 1;255,175
```

This will draw a fairly decent line, even though it has gaps in it wherever it hits some writing. Now type in exactly the same command again. The line will vanish without leaving any trace whatsoever - this is the great advantage of OVER 1. If you had drawn the line using...

```
PLOT 0,0: DRAW 255,175
```

...and erased it using...

```
PLOT 0,0: DRAW INVERSE 1;255,175
```

...then you would also have erased some of the writing.

Now try...

```
PLOT 0,0: DRAW OVER 1;250,175
```

...and try to undraw it using...

```
DRAW OVER 1;-250,-175
```

This doesn't quite work, because the pixels that the line uses on the way back are not quite the same as the ones that it used on the way there. You must therefore undraw a line in exactly the same direction as you drew it.

One way to get unusual colours is to speckle two normal ones together in a single square, using a user-defined graphic. Try this program...

```
1000 FOR n=0 TO 6 STEP 2
1010 POKE USR "a"+n, BIN 01010101:
      POKE USR "a"+n+1, BIN 10101010
1020 NEXT n
```

...which gives the user-defined graphic corresponding to a chessboard pattern. If you print this character (press **[GRAPH]**, then **A**) in red ink on yellow paper, you will find it gives a reasonably acceptable orange.

Exercises...

1. Experiment with **PAPER**, **INK**, **FLASH** and **BRIGHT** items in a **PLOT** statement. These are the parts that affect the whole of the character cell containing the pixel. Normally it is as though the **PLOT** statement had started off...

```
PLOT PAPER 8; FLASH 8; BRIGHT 8; ...etc...
```

...and only the ink colour of a character cell is altered when something is plotted there, but you can change this if you wish.

Be especially careful when using colours with **INVERSE 1**, because this sets the pixel to show the paper colour, and may change the ink colour, which might not be what you expect.

2. Try to draw circles using **SIN** and **COS** (if you have read part 10, see if you can work out how). Run this..

```
10 FOR n=0 TO 2*PI STEP PI/180
20 PLOT 100+80*COS n,87+80*SIN n
30 NEXT n
40 CIRCLE 150,87,80
```

You can see that the **CIRCLE** statement is much quicker, albeit less accurate.

3. Try...

```
CIRCLE 100,87,80: DRAW 50,50
```

You can see from this that the **CIRCLE** statement leaves the **PLOT** position at a rather indeterminate place - it is always somewhere about half way up the right hand side of the circle. You will usually need to follow the **CIRCLE** statement with a **PLOT** statement before you do any more drawing.

Part 18

Motion

Subjects covered...

PAUSE, INKEY\$, PEEK

Quite often you will want to make the program take a specified length of time, and for this you will find the PAUSE statement useful.

PAUSE n

...stops computing and displays the picture for n frames of the TV (at 50 frames per second in Europe or 60 in USA). The value of n can be up to 65535, which gives you a pause of just under 22 minutes. If $n=0$ then it means 'pause indefinitely'.

A pause can always be cut short by pressing a key.

This program works the second hand of a clock...

```
10 REM first we draw the clock face
20 FOR n=1 TO 12
30 PRINT AT 10-10*COS (n/6*PI),16+10*SIN (n/6*PI);n
40 NEXT n
50 REM now we start the clock
60 FOR t=0 TO 200000: REM t is the time in seconds
70 LET a=t/30*PI: REM a is the angle of the second hand in
  radians
80 LET sx=80*SIN a: LET sy=80*COS a
200 PLOT 128,88: DRAW OVER 1;sx,sy: REM draw second hand
210 PAUSE 42
220 PLOT 128,88: DRAW OVER 1;sx,sy: REM erase second hand
400 NEXT t
```

The clock will run down after about 55.5 hours because of line 60, but you can easily make it run longer. Note how the timing is controlled by line 210. You might expect PAUSE 50 to make it tick once per second, however, the computing takes a bit of time as well and has to be allowed for. This is best done by trial and error, timing the +2 clock against a real one, and adjusting line 210 until they agree. (You can't do this very accurately - an adjustment of one frame per second is equal to 2% or half an hour in a day.)

There is a much more accurate way of measuring time. This uses the contents of certain memory locations. The data stored is retrieved by using PEEK. Part 25 of this chapter explains what we're looking at in detail. The expression used is...

$(65536*PEEK\ 23674+256*PEEK\ 23673+PEEK\ 23672)/50$

This prints the number of seconds since the **+2** was switched on or **RESET** (up to about 3 days and 21 hours, after which it goes back to 0).

Here is a revised clock program to make use of this...

```
10 REM first we draw the clock face
20 FOR n=1 TO 12
30 PRINT AT 10-10*COS (n/6*PI),16+10*SIN (n/6*PI);n
40 NEXT n
50 DEF FN t()=INT ((65536*PEEK 23674+256*PEEK 23673+
  PEEK 23672)/50): REM number of seconds since start
100 REM now we start the clock
110 LET t1=FN t()
120 LET a=t1/30*PI: REM a is the angle of the second hand in
  radians
130 LET sx=72*SIN a: LET sy=72*COS a
140 PLOT 131,91: DRAW OVER 1;sx,sy: REM draw hand
200 LET t=FN t()
210 IF t<=t1 THEN GO TO 200: REM will wait until time for
  next hand
220 PLOT 131,91: DRAW OVER 1;sx,sy: REM rub out old hand
230 LET t1=t: GO TO 120
```

The internal clock that this method uses should be accurate to about 0.01% (approx 10 seconds per day) so long as the **+2** is simply running the program. However, when you use the **BEEP** statement (described in part 19 of this chapter) or operate the datacorder, printer or any other peripheral attached to the **+2**, the internal clock stops temporarily, losing time.

The numbers **PEEK 23674**, **PEEK 23673** and **PEEK 23672** are held inside the **+2** and used for counting in 50ths of a second. Each is between 0 and 255 and they gradually increase through all the numbers from 0 to 255; after 255 they drop straight back to 0.

The one that increases most often is **PEEK 23672** - every 1/50 second it increases by 1. When it is at 255, the next increase 'nudges' it to 0, and at the same time it increments **PEEK 23673** up by 1. When (every 256/50 seconds) **PEEK 23673** is nudged from 255 to 0, it in turn increments **PEEK 23674** up by 1. This should be enough to explain why the expression above works.

Now, consider carefully - suppose our three numbers are 0 (for **PEEK 23674**), 255 (for **PEEK 23673**) and 255 (for **PEEK 23672**). This means that it is about 21 minutes after switch on. Our expression ought to yield...

$$(65536*0+256*255+255)/50=1310.7$$

But there is a hidden danger - the next time there is a 1/50 second count, the three numbers will change to 1, 0 and 0. Every so often, this will happen when you are half way through evaluating the expression - the **+2** would evaluate **PEEK 23674** as 0, but then change the other two to 0 before it can **PEEK** them. The answer would then be...

$$(65536*0+256*0+0)/50=0$$

...which is obviously wrong.

A simple way of avoiding this problem is to evaluate the expression *twice in succession* and take the larger answer.

If you look carefully at the previous program, you can see that it does this implicitly.

Here is a trick to apply the rule. Define the functions...

```
10 DEF FN m(x,y)=(x+y+ABS (x-y))/2: REM the larger of
   x and y
20 DEF FN u()=(65536*PEEK 23674+256*PEEK 23673+PEEK
   23672)/50: REM time (may be wrong)
30 DEF FN t()=FN m(FN u(), FN u()): REM time (correct)
```

You can change the three counter numbers so that they give the real time instead of the time since the **+2** was switched on. For instance, to set the time at 10.00am, you work out that this is $10 \times 60 \times 60 \times 50 = 1800000$ fiftieths of a second, and that...

$$1800000 = 65536 * 27 + 256 * 119 + 64$$

To set the three numbers to 27, 119 and 64, you type...

```
POKE 23674,27: POKE 23673,119: POKE 23672,64
```

In countries with mains frequencies of 60 Hz (cycles per second), these programs must replace '50' by '60' where appropriate.

The function `INKEY$` (which has no argument) reads the keyboard. If you are pressing just one key, (or say, **[CAPS SHIFT]** and just one other key), then the result is the character which that key gives normally, otherwise the result is an empty string.

Try this program, which works like a typewriter.

```
10 IF INKEY$ <> "" THEN GO TO 10
20 IF INKEY$ = "" THEN GO TO 20
30 PRINT INKEY$;
40 GO TO 10
```

Here line 10 waits for you to lift your finger off the keyboard, and line 20 waits for you to press a new key.

Remember that unlike `INPUT`, `INKEY$` doesn't wait for you, so you don't have to press **[ENTER]**. On the other hand, if you don't type anything at all, then you've missed your chance.

Exercises...

1. What happens if you miss out line 10 in the 'typewriter' program?

2. Another way of using `INKEY$` is in conjunction with `PAUSE` as in this alternative typewriter program...

```
1Ø PAUSE Ø  
2Ø PRINT INKEY$;  
3Ø GO TO 1Ø
```

To make this work, why is it essential that a pause should not finish if it finds you already pressing a key when it starts?

3. Adapt the 'second hand' program so that it also shows minute and hour hands, re-drawing them every minute. If you're feeling ambitious, arrange so that every quarter of an hour it puts on some kind of 'show' - perhaps you could produce the 'Big Ben' chimes using `PLAY` (described next in part 19 of this chapter).

Part 19

Sound

Subjects covered...

BEEP, PLAY

As you will have already noticed, the +2 can make a variety of noises. To get the best quality of sound, it's important to make sure that your TV is tuned in properly (see chapter 2). If, instead of a TV, you are using a VDU monitor (which won't reproduce the +2's sound), note that a separate sound signal (which may be connected to an audio amplifier powering speaker(s) or headphones) is available from the **SOUND** socket at the back the +2. Headphones may *not* be plugged into the **SOUND** socket directly.

Connections to the **SOUND** socket are described in chapter 10.

To get the most out of the +2's musical ability, it helps to have a little knowledge about musical terms.

Note - In the examples that follow, it is important that you type in the string expressions *exactly* as shown in upper case and lower case letters, ie. the example "g a" should *not* be typed in as "G a", "g A" or "G A".

Type in this command (don't worry about what it means just yet)...

```
PLAY "g a"
```

Two notes were played - the second slightly higher than the first. The difference between the notes is called a *tone*. Now try...

```
PLAY "g$a"
```

Again there were two notes played - the first one was the same as the previous example, but there was less of a jump to the second. If you didn't hear the difference, then try the first example followed by the second again. The second example has half the difference between notes, and this is called a *semitone*.

When you're happy with semitones, try this...

```
PLAY "gD"
```

This sort of difference is called a *fifth*, and occurs quite often in music of all kinds. With that example ringing in your ears, type...

```
PLAY "gG"
```

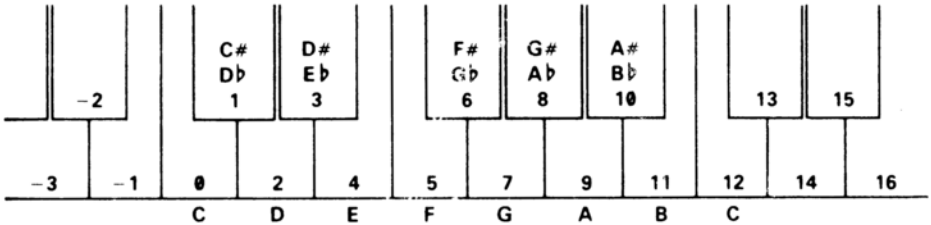
Although (hopefully) you noticed that there was a much bigger difference that time than for the fifth, the two notes somehow sounded much more similar. This is called an *octave*, and is the point at which music starts to 'repeat itself'. Don't worry about that unduly, just remember what an octave sounds like.

There are *two* ways of making music and sounds with the `+2`. The most elementary is the somewhat spartan `BEEP` command. This takes the form...

`BEEP duration , pitch`

...where, as usual, *duration* and *pitch* represent numerical expressions. The duration is given in seconds, and the pitch is given in semitones above middle C - using negative numbers for notes below middle C.

Here is a diagram to show the pitch values of all the notes in one octave on the piano for `BEEP`...



Hence, to play the A above middle C for half a second, you would use...

`BEEP 0.5,9`

...and to play a scale (for example, C major) a complete (albeit short) program is needed...

```

10 FOR f=1 to 8
20 READ note
30 BEEP 0.5,note
40 NEXT f
50 DATA 0,2,4,5,7,9,11,12

```

To get higher or lower notes, you have to add or subtract 12 for each octave that you go up or down.

`BEEP` exists mostly to provide compatibility with the older designs of Spectrum, though it can be useful for very short or rapid sound effects. For any new programs you develop, the second way of producing sound is much to be preferred, and this is called `PLAY` (if you worked through the simple examples earlier in this section, you'll remember that that's what you used).

PLAY is much more flexible than BEEP - it can play up to three voices in harmony with all manner of effects, and gives a much higher quality of sound. It's also much easier to use. For example, to play A above middle C for half a second, type in...

```
PLAY "a"
```

...and to play the C major scale (which needed a program to itself before), use...

```
PLAY "c d e f g a b C"
```

Notice that the last C in the example above is in upper case. This tells the PLAY command to play it an octave higher than the lower case c. A *scale*, by the way, is the term used for a set of notes spanning an octave. The example above is called the C major scale because it's the set of notes between two C's. Why major? There are two main classes of scale, major and minor, and this is just musical shorthand for describing two different sets. Just for interest, the C minor scale sounds like this...

```
PLAY "c d $ e f g $ a $ b C"
```

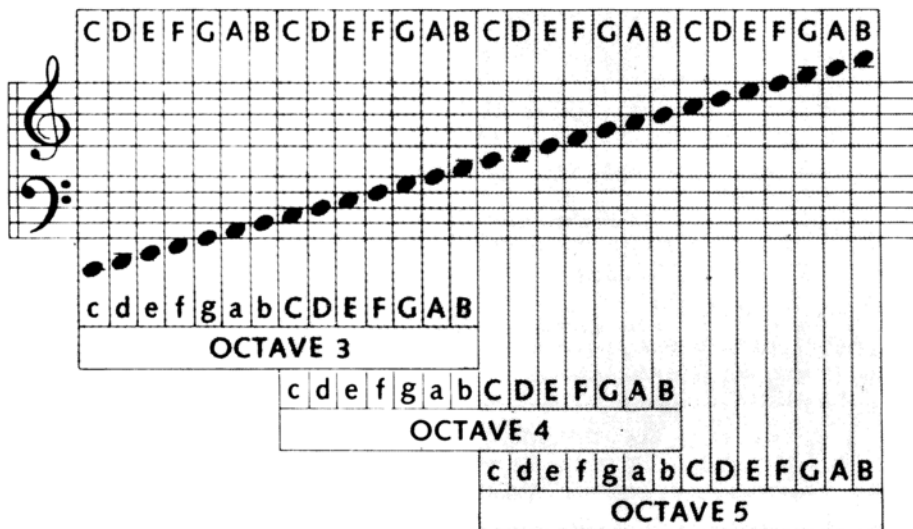
Preceding a note by \$ drops it by a semitone (*flattens* it), and preceding a note by # raises it by a semitone (*sharpens* it). The PLAY command spans 9 octaves, and can be told which one to use by having the upper case letter O followed by a number, in the list of notes it is given. Type in this little program...

```
10 LET o$="05"  
20 LET n$="DECcg"  
30 LET a$=o$+n$  
40 PLAY a$
```

There are a few new things in this program. Firstly, PLAY is just as happy with a string variable as with a string constant. In other words, providing that a \$ has been set up beforehand, PLAY a\$ works just as well as PLAY "05DECcg". In fact, using variables in PLAY statements has certain distinct advantages, and we shall do this from now on.

Notice also that the string a\$ has been 'built up' by combining two smaller strings o\$ and n\$. While this doesn't make much difference at this sort of level, PLAY can cope with strings many thousands of notes long, and the only sensible way of creating and editing those strings from BASIC is to combine lots of smaller strings in this way.

Now run the above program. Edit line 10 so that "05" becomes "07", and run it again, or if you want to be a big spaceship make it "02". If you don't specify an octave number for a particular string, then the +2 assumes that you want octave 5. Here is a diagram of the notes and octave numbers which correspond to the standard even-tempered musical scale.












There is a lot of overlap, so for example, "03D" is the same as "04d". This makes it easier to write tunes without having to change octave all the time. Some of the notes in the lowest octaves (0 and 1) aren't very accurate for technical reasons, and so the computer just makes a brave attempt at getting as close as possible.




PLAY can also handle many different lengths of note. Edit the program above so that line 10 is now...

```
10 LET o$="2"
```

...and run it. Then alter the setting of o\$ between "1" and "9". The note length can be changed anywhere in a string by including a number between 1 and 9, and this is effective for all subsequent notes until a new number is encountered. Each of these nine note lengths has a specific musical name, and looks different when written down in musical notation. The following table shows which is which...

NUMBER	NOTE NAME	MUSICAL SYMBOL
1	semi-quaver	
2	dotted semi-quaver	
3	quaver	
4	dotted quaver	
5	crotchet	
6	dotted crotchet	
7	minim	
8	dotted minim	
9	semi-breve	

PLAY can also cope with *triplets* , which are three notes played in the time for two. Unlike simple note lengths, the triplet number only applies for the three notes immediately following, and then the previous note length number resumes. The triplet numbers are as follows...

NUMBER	NOTE NAME	MUSICAL SYMBOL
10	triplet semi-quaver	
11	triplet quaver	
12	triplet crotchet	

PLAY is quite happy about being told to 'shut up'! A timed period during which no notes play is called a *rest* , and "&" is used to signify this. The length of rest it produces is the same as the current note length. To demonstrate, edit lines 10 and 20 to...

```
10 LET o$="04"
20 LET n$="DEC&c g"
```

Two notes played together without a break are called *tied notes* , which are signified in a PLAY command by an underline, so a crotchet c and a minim c tied together would be "5_7 c". (The second value is then used as the note length for all subsequent notes, as before.)

There are occasions when ambiguity creeps in. Say that a piece of music needs octave 6 and a note length of 2, then...

```
10 LET o$="062"
```

...seems a good bet - but no! The computer will find the 0 and try to read the number following it. When it finds 62, it will stop with the report `n Out of range`. In cases like this, there is a 'dummy note' called N that just serves to split things up, so line 10 should be...

```
10 LET o$="06N2"
```

The volume can be set between 0 (minimum) and 15 (maximum) using "V" followed by a number. In practice, only 10 to 15 are likely to be useful, as 1-9 are too soft unless the +2 is being used with an amplifier. As previously mentioned, BEEP is louder than a single channel of PLAY, but if all three channels play a note at volume 15, then it should be at the same level as a note produced by BEEP.

Playing more than one channel at a time is very simple; you just separate lists of notes by commas. Try this new program...

```
10 LET a$="04cCcCgGgG"
20 LET b$="06CaCe$bd$bD"
30 PLAY a$,b$
```

In general, there is no difference between the three channels, and any string of notes can be put onto any channel. The overall speed of the music, the *tempo*, must be in the string assigned to channel A (the first string after PLAY), otherwise it will be ignored. To set tempo in beats (crotchets) per minute, use "T" followed by a number between 60 and 240. The standard value is 120, or two crotchets per second. Modify the program above to...

```
5 LET t$="T120"
10 LET a$=t$+"04cCcCgGgG"
20 LET b$="06CaCe$bd$bD"
30 PLAY a$,b$
```

...and run it several times, changing line 5 for different tempos

A common feature in music is the repetition of a group of notes. Any part of a string can be repeated by enclosing it in brackets, so if you change line 10 to...

```
10 LET a$=t$+"04(cC)(gG)"
```

PLAY treats it just the same as the old line 10. If you include a closing bracket, (with no matching opening bracket) then the string up to that point is repeated indefinitely. This is useful for rhythm effects and bass lines. To demonstrate, try this (you'll have to use [BREAK] to stop the sound)...

```
PLAY "04N2cdefgfed)"
```

...and...

```
PLAY "04N2cd(efgf)ed)"
```

If you set up an infinitely repeating bass line, and then play a melody with it, then it would be nice if the bass line stops when the melody does. There is a device to do this - if PLAY comes across "H" (for Halt) in any of the strings it is playing, then it stops all sound immediately. Run the following program (again, you'll have to use [BREAK] to stop it)...

```

10 LET a$="cegbdfaC"
20 LET b$="04cC)"
30 PLAY a$,b$

```

Now modify line 10 to...

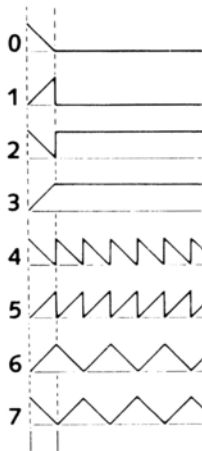
```

10 LET a$="cegbdfaCH"

```

...and run it again.

So far we've only used notes which start and stop at one level of volume. The **+2** can alter the volume of a note while it is playing, so it can start loud and die away like a piano, or rise and fall like a dog growling. To turn these effects on, use "W" (for Waveform) followed by a number between 0 and 7 together with "U" for each channel you want to use the effect on. Any channel with a volume setting ("V") will not respond to "U". This table shows graphically how the volume changes for each setting...



- 0 - single decay then off.
- 1 - single attack then off.
- 2 - single decay then hold.
- 3 - single attack then hold.
- 4 - repeated decay.
- 5 - repeated attack.
- 6 - repeated attack-decay.
- 7 - repeated decay-attack.

This program plays the same note with each effect in turn, so you can compare them against the diagram above.

```
10 LET a$="UX1000W0C&W1C&W2C&W3C&W4C&W5C&W6C&W7C"
20 PLAY a$
```

The U turns on effects, and the W selects which waveform to use. There's also an "X1000". X sets how long the effect will last for (from 0 to 65535). If you don't include an X, then the +2 will choose the longest value. Waveforms that settle down (0 to 3 in the table above) after the initial part, work best with X settings of about 1000, whereas repetitive effects (4-7) are more effective with short values like 300. Try varying the X setting in the program above to get some idea of how each works.

The PLAY command isn't limited to pure musical notes. There are also three 'white noise' generators (white noise is a sound which is like an un-tuned FM radio or TV), and any of the three channels can play notes, white noise, or a mixture of both. To select a mix of noise and note, you may use 'M' followed by a number between 1 and 63. You can work out which number to use from this table...

	Tone channels			Noise channels		
	A	B	C	A	B	C
Number	1	2	4	8	16	32

Write down the numbers corresponding to the effects you want, and then add them together. If you wanted A to be noise, B to be tone, and C to be both tone and noise, then add 8, 2, 4 and 32 to get 46 (the order of the channels is the order of the strings which follow the PLAY command). The best effects can be obtained with the A channel - don't be afraid to experiment.

By now, you'll be writing symphonies. However, it can be difficult to work out just which part of the music a particular section of string is responsible for. To alleviate this problem, your music string may include 'comments' enclosed between ! exclamation marks; for example...

```
1098 LET z$=z$+"CDcE3Ge4_6f! end of 75th bar !egeA"
```

The PLAY command will simply 'hop over' any comments in the string.

If you have an electronic musical instrument with MIDI, then the +2 can control it using PLAY. Up to 8 channels of music can be sent to synthesisers, drum machines or sequencers. The PLAY command is constructed exactly as described so far in this section, except that each string should include a "Y" followed by a number between 1 and 16. The number after the Y controls which channel the music data is assigned to. Up to eight strings can be used; the first three strings will still be played through the TV as before so you'll probably want to turn the TV sound down. You can also send MIDI programming codes via the PLAY command, using "Z" followed by the code number. Key velocities (loudness) are calculated and sent at 8 times the V setting (so "V6" will send 48 as a key velocity).

Finally, here is a brief list of the parameters that can be used in string of a PLAY command, together with the values they may take...

STRING	FUNCTION
a - g A - G	Specifies the pitch of the note within the current octave range.
\$	Specifies that the note which follows must be flattened.
#	Specifies that the note which follows must be sharpened.
0	Specifies the octave number to be used (followed by 0 - 8).
1 - 12	Specifies the length of notes to be used.
&	Specifies that a rest is to be played.
-	Specifies that a tied note is to be played.
N	Separates two numbers.
V	Specifies the volume to be used (followed by 0 - 15).
W	Specifies the volume effect to be used (followed by 0 - 7).
U	Specifies that the volume effect to be used in a string.
X	Specifies duration of volume effect (followed by 0 - 65535).
T	Specifies tempo of music (followed by 60 - 240).
()	Specifies that enclosed phrase must be repeated.
! !	Specifies that enclosed comment is to be skipped over.
H	Specifies that the P L A Y command must stop.
M	Specifies the channel(s) to be used (followed by 1 - 63).
Y	Specifies that MIDI channel is to be used (followed by 1 - 16).
Z	Specifies MIDI programming code (followed by code number).

Part 20

Datascorder operations

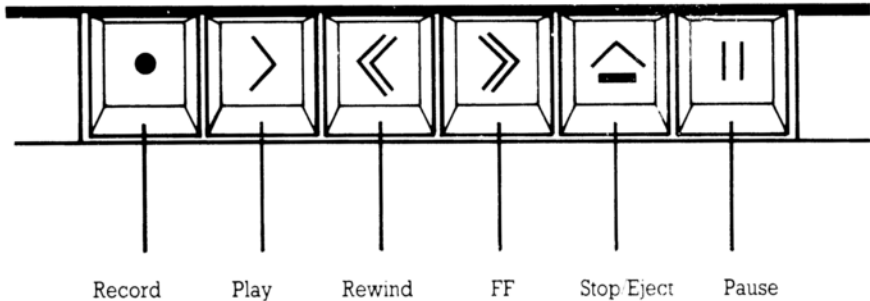
Subjects covered...

LOAD, SAVE, VERIFY, MERGE

The basic method of using the datascorder to load software is given in chapters 3 and 4.

You can also use the datascorder to store (*save*) your own programs onto cassette so that you can load them back into the computer whenever you wish to use them - (otherwise, you would always need to type in every program from scratch).

First of all, familiarise yourself with the datascorder's six function keys...



To see how the datascorder saves a program, first type in the short program (which displays coloured squares) that you first met at the end of part 16, i.e....

```
10 POKE 22527+RND*704, RND*127
20 GO TO 10
```

This is the program that you are going to save onto cassette. Any standard cassette should work although low noise cassettes may be better.

Type in the following...

```
SAVE "squares"
```

"squares" is just a name that you use to 'label' the program you are going to store on cassette. You are allowed up to ten characters in the name.

The +2 will display the message...

```
Press REC & PLAY, then any key.
```

We shall first go through a 'dry run' so that you can see what will happen when we actually *do* save the program later. This time, therefore, *don't* press record and play on the datacoder - just press a key on the +2 (for example [ENTER]) and watch the border of the TV screen. You will see patterns of coloured horizontal stripes as follows:

Five seconds of red and cyan stripes moving slowly upwards, followed by a very short burst of blue and yellow stripes.

A short pause.

Two seconds of the red and cyan stripes again, followed by another short burst of blue and yellow stripes.

While the stripes appear on the screen, you can also hear the 'sound' of the data through your TV's speaker.

Keep trying out the above SAVE command (without actually operating the datacoder) until you can recognise these patterns. What's actually happening is that the information is being saved in two *blocks* and both blocks have a 'lead-in' (which corresponds to the red and cyan stripes) followed by the information itself (which corresponds to the blue and yellow stripes). The first block is a preliminary one containing the name and various other bits of information about the program, and the second is the program itself together with any variables present. The pause between them is just a gap.

Now let's actually save the program onto cassette:

1. Wind the cassette to an area of tape that is either blank, or that you are prepared to overwrite.
2. Type...

```
SAVE "squares"
```

3. Obey the message 'Press REC & PLAY, then any key.'
4. Watch the screen as before. When the +2 has finished (with the report 'Ø OK') stop the datacoder.

When you have *successfully* saved a program, you can happily switch off or reset the computer, or start a NEW program, knowing that you could always load in the saved program if you needed it. However, before clearing the saved program from the computer's memory, you should always make sure that the save worked correctly - you can check the signal on the cassette against the program in the memory using the VERIFY command:

1. Rewind the cassette to just before the point at which you saved the program.
2. Type...

```
VERIFY "squares"
```

The border will alternate between red and cyan until the **+2** finds the program you specified, then you will see the same pattern as you did when you saved the program. During the pause between the blocks, the message 'Program: squares' will be displayed on the screen. (When the **+2** is searching for something on cassette, it displays the name of everything it comes across.) If, after the pattern has appeared, the computer displays the report 'Ø OK', then your program is safely stored on cassette and you can skip the next five paragraphs. Otherwise, something has gone wrong - take the following steps to find out what:

If the program name has not been displayed, then either the program was not saved properly in the first place, or it was, but was not 'read back' properly. You need to find out which. To see if it was saved properly, rewind the cassette to just before where you saved the program, and play it back while listening to the TV's speaker. The (red and cyan) lead-in should produce a clear, steady high pitched note, and the (blue and yellow) information part gives a much less pleasant screech.

If you do not hear these noises, then the program was probably not saved. Check that you were not trying to save the program onto the plastic leader at the beginning of the cassette. When you have checked this, try saving again.

If you can hear the sounds as described, then SAVE was probably alright and your problem is with reading back.

It could be that you mistyped the program name when you *saved* it (in which case when the **+2** finds the program it will display the mistyped name on the screen). On the other hand, perhaps you mistyped the program name when you *verified* it, in which case the computer will ignore the correctly saved program and carry on looking for the wrong name, flashing red and cyan as it goes.

If there is a genuine mistake on the cassette, then the **+2** will display the message 'R Tape loading error' which means in this case that it failed to verify the program. Note that a slight fault on the tape itself (which might be almost inaudible with music) can wreak havoc with a computer program. Try saving the program again, perhaps on a different part of the tape.

Now let us suppose that you have saved the program and successfully verified it. Loading it back into the memory is just like verifying it except that you type...

```
LOAD "squares"
```

...(instead of VERIFY "squares").

Since the program verified properly, you should have no problem loading it.

LOAD deletes the old program (and variables) in the memory when it loads in the new one from cassette.

Once a program has been loaded, the command RUN will run it.

As mentioned in chapters 3 and 4, it is possible to buy pre-recorded programs (software) on cassette. They must be specially written for the ZX Spectrum range (ie. The Spectrum, the Spectrum +, the Spectrum 128 or the Spectrum **+2**). Different makes and models of computer have different ways of storing programs, so they *cannot* use each other's cassettes.

If your cassette has more than one program stored on the same side, then each will have a name. You can choose which program to load in the `LOAD` command - for instance, if the one you want is called 'helicopter', you could type...

```
LOAD "helicopter"
```

The command `LOAD ""` means load the first program that the computer comes across on the cassette. This can be very useful if you cannot remember the name that you saved the program under!

The option 'Tape Loader' from the opening menu has the same action as `LOAD ""`, and is much quicker to use - simply switch on the **+2** and press **[ENTER]**.

As previously mentioned, `LOAD` deletes the old program and variables in the computer whenever it loads in the new ones from cassette; however, there is another command, `MERGE`, which is similar to `LOAD` but it only deletes an old program line or variable if there is a new one with the same line number or name. Type in the 'dice' program in part 11 of this chapter and `SAVE` it onto cassette, as "dice". Now enter and run the following new program...

```
1 PRINT 1
2 PRINT 2
10 PRINT 10
20 LET x=20
```

Rewind the cassette so that you are ready to load in the dice program, then type in...

```
MERGE "dice"
```

And follow the same procedure as if you were `LOAD`ing the program using the datacorder. If you then `LIST` the program, you will see that lines 1 and 2 have survived, but lines 10 and 20 have been overwritten by those from the dice program. The variable `x` has also survived (try `PRINT x`).

You have now seen simple forms of the four commands that work in conjunction with the datacorder:

- `SAVE` - Stores the program and variables on to cassette.
- `VERIFY` - Checks the program and variables on cassette against those in the computer's memory.
- `LOAD` - Clears the computer of all its program and variables, and replaces them with new ones read in from cassette.
- `MERGE` - Similar to `LOAD` except that it does *not* clear the old program lines and variables unless it *has to* (because they are the same as those being loaded in from cassette).

In each of the above commands, the keyword is followed by a string. For the `SAVE` command, this string consists of a name by which the program is stored on cassette, while for the other three commands, the string tells the computer which program to search for. While the computer is searching, it displays the name of each program it comes across. There are a couple of twists to all this, however:

For `VERIFY`, `LOAD` and `MERGE` you can provide the empty string `""` as the name to search for; then the computer does not care about the name, but takes the first program it comes across.

A variant on SAVE takes the form...

```
SAVE string LINE number
```

A program which saved using this command, is stored in such a way that when it is read back by LOAD (but not MERGE) it automatically jumps to the line with the given number, thus running itself.

If you load a program which *doesn't* automatically run (by using the 'Tape Loader' option from the opening menu), then you'll have to select the '128 BASIC' option after the program has loaded, in order to run it or edit it.

So far, the only kinds of information we have stored on cassette have been programs together with their variables. There are two other kinds as well, called *arrays* and *bytes*.

You can save arrays on cassette using the keyword DATA in a SAVE statement...

```
SAVE string DATA array name ( )
```

...where *string* is the name that the information will have on cassette and works in exactly the same way as when you save a program (or plain bytes).

The *array name* specifies the array you want to save, so it is just a letter (or a letter followed by \$). Remember to put the brackets () after the array name.

Be clear about the separate roles of *string* and *array name*. If you say (for instance)...

```
SAVE "Bloggs" DATA b( )
```

...then SAVE takes the array *b* from the computer and stores it on cassette under the name "Bloggs".

When you type...

```
VERIFY "Bloggs" DATA b( )
```

...the computer will look for a number array stored on cassette under the name "Bloggs". When it finds one, it will display 'Number array: Bloggs' and check it against the array *b* in the computer.

The command...

```
LOAD "Bloggs" DATA b( )
```

...finds the array on cassette, and then (if there is room for it in the computer) deletes any array already existing called *b* and loads in the new array from cassette, calling it *b*.

You *cannot* use MERGE with saved arrays.

You can save character (string) arrays in exactly the same way. When the computer is searching the cassette and finds one of these it writes up 'Character array:' followed by the name. When you load in a character array, it will delete not only any previous character array with the same name, but also any simple string variable with the same name.

Byte storage is used for pieces of information without any reference to what the information is used for - it could be a TV screen display, or perhaps some user-defined graphics, or just something you have made up for yourself. It is specified using the word CODE, as in...

```
SAVE "picture" CODE 16384,6912
```

The unit of storage in memory is the *byte* (a number between 0 and 255), and each byte has an *address* (which is a number between 0 and 65535). The first number after CODE is the address of the first byte to be stored on cassette; the second number is the *amount* of bytes to be stored. In our case, 16384 is the address of the first byte in the file (which contains the TV screen display), and 6912 is the amount of bytes in it, so we are saving a actual copy of the TV screen. Try the above SAVE command. (You don't have to save the bytes using the name "picture" - it's merely a convenient reminder of what's on the cassette.)

To load it back, use...

```
LOAD "picture" CODE
```

You can put numbers after CODE in the form...

```
LOAD name CODE start ,length
```

Here, *length* is used as a safety measure - when the computer has found the bytes on cassette with the right name, it will check the *length* and refuse to load the bytes if there are more than specified (thereby safeguarding against the extra bytes accidentally overwriting an area of memory you wished to preserve). In such a case, the report 'R Tape loading error' is also displayed.

If you leave out *length*, the computer will read in the bytes however many there are.

The *start* parameter shows the address where the first byte is to be loaded back to - this can be different from the address it was saved from, although if they are the same, then you can leave out *start* in the LOAD statement.

CODE 16384,6912 is such a useful area of memory (the screen display) to save and load, that a special function (SCREEN\$) has been provided to represent it, so you can type (for example)...

```
SAVE "picture" SCREEN$
```

...or...

```
LOAD "picture" SCREEN$
```

This is one of the rare cases where VERIFY will *not* work - VERIFY displays the names of what it finds on cassette, thereby altering the saved screen display as it does so, and therefore the verification fails.

Anything you can do with SAVE, LOAD or MERGE on cassette, you can also do with the *silicon disc* that's built into the +2. This acts like a cassette (with a couple of extra commands), with the exception that it's about 64K in size, very fast and loses its contents when the +2 is reset or turned off (however, it *does* survive the NEW command). You use all the commands in exactly the same way you would with the datacorder - simply add an exclamation mark ! between the command and its associated string. So where you would type...

```
SAVE "squares"
```

...to save to cassette, you may instead use...

```
SAVE ! "squares"
```

...to save to the silicon disc.

There are two extra commands for use with silicon disc. The first one is...

```
CAT !
```

...which gives you a list of all the programs or data that's stored in the disc.

The second command is...

```
ERASE ! "filename"
```

...to get rid of an unwanted program or data.

Perhaps the most obvious use of the silicon disc is to store chunks of BASIC program which can be merged (using MERGE !) into a smaller program, in sequence. This makes it possible to write about 90K of BASIC program, and hold it in the +2 (to do this, the program structure has to be well defined).

One of the more interesting uses of the silicon disc is in *animation*, where a series of pictures can be defined by a 'slow' BASIC program stored in silicon disc, then called back to the screen at high speed. The following program gives a faint taste of this; doubtless you can do better...

```
10 INK 5: PAPER 0: BORDER 0: CLS
20 FOR f=1 TO 10
30 CIRCLE f*20,150,f
40 SAVE ! "ball"+STR$(f) CODE 16384,2048
50 CLS
60 NEXT f
70 FOR f=1 TO 10
80 LOAD ! "ball"+STR$(f) CODE
90 NEXT f
100 BEEP 0.01, 0.01
110 FOR f=9 TO 2 STEP -1
120 LOAD ! "ball"+STR$(f) CODE
130 NEXT f
140 BEEP 0.01, 0.01
150 GO TO 70
160 REM use GO TO 160 to clear the pictures from disc
170 FOR f=10 TO 1 STEP -1
180 ERASE ! "ball"+STR$(f)
190 NEXT f
```

Note that in line 40 of this program, the two numbers following `CODE` are the address in memory of the start of the screen, and the length of the top third of it. By only saving and loading the top third, the overall speed is maintained. Lines 160 to 190 are there if you **[BREAK]** out of the program, modify the circle drawing bit and try to save a new set of pictures. So before doing that, type `GO TO 160` to clear out the silicon disc. (Always try to delete files backwards, so the last file to be saved will be the first to be deleted. This saves the computer a lot of juggling about, and is much faster.)

Finally in this section, here is a complete summary of the four datacorder statements:

The parameter *name* stands for any string expression, and refers to the name under which the information is saved on cassette. It should consist of ASCII printing characters, of which only the first 10 are used.

There are four sorts of information that can be stored on cassette or silicon disc: program and variables (together), number arrays, character arrays, and bytes.

When `VERIFY`, `LOAD` and `MERGE` are searching the cassette for information with a given name and of a given sort, the computer displays on the screen the *type* and *name* of all the information it finds. The type is shown by 'Program:', 'Number array:', 'Character array:', or 'Bytes:'. If *name* is an empty string (""), then the computer takes the first lot of information (of the right sort) regardless of name.

SAVE

1. Program and variables:

`SAVE (!) name LINE line number`

...saves the program and variables in such a way that `LOAD` automatically implies a '`GO TO line number`'.

2. Bytes:

`SAVE (!) name CODE start , length`

...saves *length* bytes starting at address *start*.

Note that...

`SAVE (!) name SCREEN$`

...is equivalent to...

`SAVE (!) name CODE 16384 , 6912`

...and saves the screen display.

3. Arrays:

`SAVE (!) name DATA letter ()`

...or...

`SAVE (!) name DATA letter $ ()`

...saves the numeric array whose name is *letter*, or the character array whose name is *letter \$*.

VERIFY

1. Program and variables:

VERIFY *name*

Checks the program and variables saved under *name* on cassette against those in the memory.

2. Bytes:

VERIFY *name* CODE *start* , *length*

If the bytes saved under *name* are no longer than *length*, then checks the bytes on cassette against those in memory, starting at address *start*.

VERIFY *name* CODE

...checks the bytes saved under *name* on cassette against those in memory starting at the address from which the first cassette byte was saved.

3. Arrays:

VERIFY *name* DATA *letter* ()

...OR...

VERIFY *name* DATA *letter* \$ ()

...checks the numeric array whose name is *letter*, or the character array whose name is *letter* \$, against the array *letter* or *letter* \$ in memory.

LOAD

1. Program and variables:

LOAD (!) *name*

...deletes the old program and variables, and loads in the program and variables saved under *name* from cassette. If the program was saved using SAVE *name* LINE *line number*, then LOAD it performs an automatic 'GO TO *line number*' after the program is loaded.

If the load is not successful, then the old program and variables are not deleted.

2. Bytes:

LOAD (!) *name* CODE *start* , *length*

If the bytes saved under *name* are not longer than *length*, then load the bytes from cassette into memory, starting at address *start* and overwriting whatever was there previously.

LOAD (!) *name* CODE *start*

Unconditionally load the bytes saved under *name* from cassette into memory, starting at address *start* and overwriting whatever was there previously.

LOAD (!)name CODE

...loads the bytes saved under *name* from cassette into memory starting at the address from which the first cassette byte was saved, and overwriting the bytes that were in that section of the memory before.

3. Arrays:

LOAD (!)name DATA letter ()

...or...

LOAD (!)name DATA letter \$ ()

...deletes any numeric array already called *letter*, or any character array called *letter* \$, and forms a new one from the array stored on cassette.

MERGE

1. Program and variables:

MERGE (!)name

...merges the program saved under *name* in with the one already in memory, overwriting only the program lines or variables in the old program whose line numbers or names conflict with those in the new program.

2. Bytes:

Not possible.

3. Arrays:

Not possible.

Exercise...

1. Practise saving, loading and merging programs and data onto both cassette and the silicon disc.

Part 21

Printer operations

Subjects covered...

LPRINT, LLIST, COPY

The **+2** comes with a serial port and built-in software enabling you to use a printer. These features are usable only in 128 BASIC mode.

The printer must have an RS232 (serial) interface, and if you want to produce pictures of the screen it must have an *Epson compatible quadruple-density bit-image graphics mode*.

Make sure you have the correct lead to connect the printer to the **+2** - if in doubt, consult your Sinclair dealer.

To get the **+2** and the printer communicating with each other they must both use the same *baud rate*. The baud rate is the speed at which data is transferred between computer and printer. Although it is possible that your printer can be set to different baud rates, it'll probably be easier to change the rate at the computer end. Somewhere in the printer's operating manual, the baud rate will be specified - find this out and then set the **+2** to this rate, using the command...

```
FORMAT "p"; baud rate
```

(You won't need to do this if the printer normally uses 9600 baud, as the **+2** will assume this rate by default.)

Once you have everything set up, you can use three BASIC commands to print things out. The first two, LPRINT and LLIST, are just like PRINT and LIST, except that they use the printer instead of the TV. Note that the 'Print' option from 128 BASIC's edit menu has the same effect as LLIST, but is included as an easier method of getting a listing.

Try this program for example...

```
10 PRINT "This program..."
20 LLIST
30 LPRINT "...prints out the character set, ie..."
40 FOR n=32 TO 255
50 LPRINT CHR$ n;
60 NEXT n
```

It's important to note that LPRINT and LLIST take care to screen out any embedded colour codes (and their parameters) before printing or listing anything. Embedded colour codes are a bit of a hangover from the 48K Spectrum - when included in a string they set INK, PAPER and so on. Printers on the whole tend to use these codes for completely different things like setting italics and turning on underline etc., so it would be quite dangerous to send them colour codes and hope that nothing untoward would happen. As a side effect of this, it is impossible (from BASIC) to set up any special features on a printer that use *ESCAPE* (character 27) sequences or similar control codes.

The third statement - COPY, prints out a copy of the TV screen. To demonstrate, go into the small screen, type LIST to get a listing on the screen of the program above, and then type...

COPY

The COPY command takes about 15-30 seconds to get started, so don't panic if nothing appears to happen immediately. You'll get another listing of the program on the printer, but this time it will look pretty much the same as it does on the screen. If all you get from COPY is a lot of random characters on the printer then it's likely that your printer isn't fully compatible.

You can always stop printing at any time by pressing the [BREAK] key. Some printers have what is known as a buffer, which stores text before printing. If your printer is one of these then pressing [BREAK] will not stop it immediately, although the +2 will register the break at once.

If you try and use any of the printer commands when there isn't a printer attached, then the +2 will stop dead whilst it patiently waits for the (non-existent) printer to say 'Ready'. Pressing [BREAK] will, as usual, bring the +2 back to life.

Try this...

```
10 FOR n=31 TO 0 STEP -1
20 PRINT AT 31-n,n; CHR$(CODE "0"+n);
30 NEXT n
```

You will see a pattern of characters working down diagonally from the top right-hand corner until it reaches the bottom of the screen, when the program asks if you want to scroll.

Now change AT 31-n,n in line 20 to TAB n. The program will have exactly the same effect as before.

Now change PRINT in line 20 to LPRINT. This time there will be no pause to scroll? (which does not occur with the printer).

Now change TAB n back to AT 31-n,n still using LPRINT. This time you will get just a single line of symbols. The reason for the difference is that the output from LPRINT is not printed straight away, but is stored in a buffer until either one line's-worth of printer output has accumulated, or something else 'flushes' the buffer. Hence, printing only takes place:

1. When the buffer is full.
2. After an LPRINT statement that does *not* end in a comma or semicolon.
3. When a comma, apostrophe or TAB item requires a new line.
4. At the end of a program, if there is anything left unprinted.
5. (Depending on your printer) When you set the printer off line.

Number 3 above explains why our program with TAB works the way it does. As for AT, the line number is ignored, and the LPRINT position (like the PRINT position) is moved to the column number. An AT item can never cause a line to be sent to the printer.

Exercise...

1. Make a printed graph of SIN by running the first program in part 17 of this chapter, then using COPY.

Part 22

Other peripherals

Subjects covered...

- ZX microdrives
- Network
- RS232
- Keypad

There are many peripherals (add-ons) available that you can attach to the +2 . Chapter 10 will provide you with further details regarding their connection and operation.

The ZX microdrive is a flexible high speed mass storage device. It will operate not only with SAVE, VERIFY, LOAD and MERGE, but also with PRINT, LIST, INPUT and INKEY\$.

A *network* is used for connecting several computers so that they can talk to each other - one of the uses of this is that you then need only one microdrive to serve several computers.

The RS232 interface is a standard connection that allows you to link a computer with keyboards, printers, and various other computer devices, even if they were not designed specifically for the +2 .

The keypad can be used to facilitate extra editing functions under 128 BASIC, and is also useful for fast data entry.

Part 23

IN and OUT

Subjects covered...

OUT
IN

The processor can read from (ROM and RAM) and write to (RAM) memory by using PEEK and POKE. The processor itself does not really care whether memory is ROM or RAM - it just thinks that there are 65536 memory addresses, and it can read a byte from each one (even if it's nonsense), and write a byte to each one (even if it gets lost). In a completely analogous way, there are 65536 of what are called *I/O ports* (standing for Input/Output ports). These are used by the processor for communicating with things like the keyboard or the printer, and also for controlling the extra memory and the sound chip. Some of them can be safely controlled from BASIC by using the IN function and the OUT command, but there are locations which you *must not* write to from BASIC, as you will probably cause the system to crash, losing any program and data.

IN is a function like PEEK. Its form is...

IN address

It has one argument - the port address, and its result is a byte read from that port.

OUT is a statement like POKE. Its form is...

OUT address , value

...which writes the given *value* to the port with the given *address*. How the address is interpreted depends very much upon the rest of the computer. Quite often, many different addresses will mean the same. On the +2 it is most sensible to imagine the address being written in binary, because the individual bits (each of which can have the value either 0 or 1) tend to work independently. There are 16 bits, which we shall refer to (using *A* for address) as...

A15, A14, A13, A12, A11, A10, A9, A8, A7, A6, A5, A4, A3, A2, A1, A0

Here, A0 is the 1s bit, A1 is the 2s bit, A2 is the 4s bit, and so on. Bits A0, A1, A2, A3 and A4 are the important ones. They are normally 1, but if any one of them is 0, then this tells the computer to do something specific. The computer cannot cope with more than one thing at a time, so no more than one of these five bits should be 0. Bits A6 and A7 are ignored, so if you are a wizard with electronics you can use them yourself. The best addresses to use are those that are 1 less than a multiple of 32, so that A0 to A4 are all 1. Bits A8, A9, and so on are sometimes used to give extra information, and are used mostly for the extra memory and sound.

The byte being written or read has 8 bits, and these are often referred to (using *D* for data) as...

D7, D6, D5, D4, D3, D2, D1, D0

Here follows a list of the port addresses used:

There is a set of input addresses that read the keyboard and the datacoder.

The keyboard is divided up into 8 half-rows of 5 keys each, viz:

IN 65278 reads the half-row **[CAPS SHIFT]** to **V**
IN 65022 reads the half-row **A** to **G**
IN 64510 reads the half-row **Q** to **T**
IN 63486 reads the half-row **1** to **5** (and **JOYSTICK 2**)
IN 61438 reads the half-row **0** to **6** (and **JOYSTICK 1**)
IN 57342 reads the half-row **P** to **Y**
IN 49150 reads the half-row **[ENTER]** to **H**
IN 32766 reads the half-row (space) to **B**

(These addresses are $254 + 256 * (255 - 2 \uparrow n)$ as n goes from 0 to 7.)

In the byte read in, bits D0 to D4 stand for the five keys in the given half-row. D0 is for the outside key, and D4 is for the one nearest the middle. The bit is 0 if the key is pressed, 1 if it is not. D6 is set by the datacoder, and is effectively random if no cassette data is present.

For **JOYSTICK 1**, bit 0 is fire, bit 1 is up, bit 2 is down, bit 3 is right and bit 4 is left. For **JOYSTICK 2**, bit 0 is left, bit 1 is right, bit 2 is down, bit 3 is up and bit 4 is fire. From BASIC, these read as the number keys.

Port address 254 in output drives the sound (D4) and the save signal to the datacoder (D3), and also sets the border colour (D2, D1 and D0).

Port addresses 254, 247 and 239 are used for the extra devices mentioned in part 22.

Port address 32765 drives the extra memory. Executing an OUT to this port from BASIC will nearly always cause the computer to crash, losing any program and data. There is a fuller description of this port in part 24 of this chapter (under the heading 'Memory management'). This port is *write only* - you cannot determine the current state of the paging by an IN instruction.

Port address 49149 drives the sound chip's data registers. Port address 65533 in output writes a register address, and in input reads a register. Judicious use of these two registers can allow sounds to be generated whilst BASIC gets on with something else, but you should be aware that they also control RS232, keypad and MIDI.

Run this program to see how the keyboard works...

```
10 FOR n=0 TO 7: REM half-row number
20 LET a=254+256*(255-2↑n)
30 PRINT AT 0,0; IN a: GO TO 30
```

...and play around by pressing keys. When you finished with each half-row, press **[BREAK]** and then type...

```
NEXT n
```

The control, data and address *busses* are all exposed at the back of the **+2** on the **EXPANSION I/O** socket, so you could do almost anything with a **+2** that you could with a Z80. Sometimes, though, the computer's hardware might get in the way.

See chapter 10 for a diagram and pin-out of the **EXPANSION I/O** socket.

Part 24

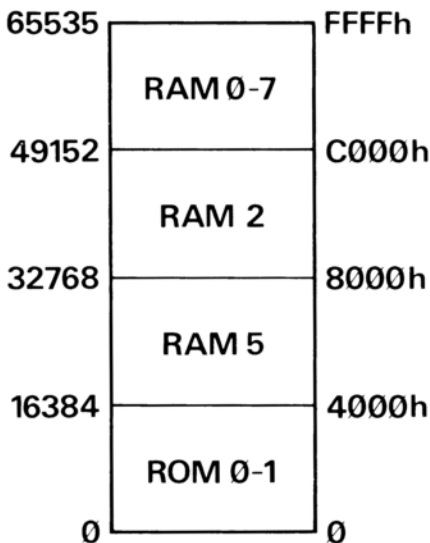
The memory

Subjects covered...

CLEAR

Deep inside the *+2*, everything is stored as *bytes*, ie. numbers between 0 and 255. You may think you have stored away the price of gruts or the address of your friend Freamsley, but in fact, all the information has been converted into collections of bytes, and bytes are what the computer sees.

Each place where a byte can be stored has an address, which is a number between 0 and FFFFh (a small *h* at the end of the digits signifies that the number is hexadecimal). This means that an address can be stored as two bytes. You might think of the memory as a long row of numbered boxes, each of which can contain a byte. Not all the boxes are the same, however - the boxes from 4000h to FFFFh are RAM boxes, which means you can open the lid and alter the contents, but those from 0 to 3FFFh are ROM boxes, which have a glass lid that cannot be opened - you just have to read whatever was put into them when the computer was made. In the *+2*, we have crammed in more than twice the amount of memory than can comfortably fit. While the processor can address 65536 bytes, there are in fact 131072 bytes of RAM and 32768 bytes of ROM making 163840 bytes (160K) in all. All this is hidden from the processor by the hardware using a process called *paging* - BASIC (and the processor) always 'sees' the memory as 16K of ROM and 48K of RAM.



The *+2* memory map

To inspect the contents of a box, we use the PEEK function. Its argument is the address of the box, and its result is the contents. For example, this program prints out the first 21 bytes in ROM (and their addresses)...

```
10 PRINT "Address"; TAB 8; "Byte"  
20 FOR a=0 TO 20  
30 PRINT a; TAB 8; PEEK a  
40 NEXT a
```

All these bytes will probably be quite meaningless to you, but the processor chip understands them to be instructions telling it what to do.

To change the contents of a box (if it is RAM), we use the POKE command. Its form is...

```
POKE address, contents
```

...where *address* and *contents* are numeric expressions. For example, if you type...

```
POKE 31000,57
```

...then the byte at address 31000 is given the new value 57. Now type...

```
PRINT PEEK 31000
```

...to prove this. (Try poking in other values, to show that there is no cheating.) The new value must be between -255 and +255; if it is negative, then 256 is added to it.

The ability to poke gives you immense power over the computer if you know how to wield it, and immense destructive possibilities if you don't. It is very easy (by poking the wrong value into the wrong address) to lose vast programs that took you hours to type in. Fortunately though, you won't do the computer any permanent damage.

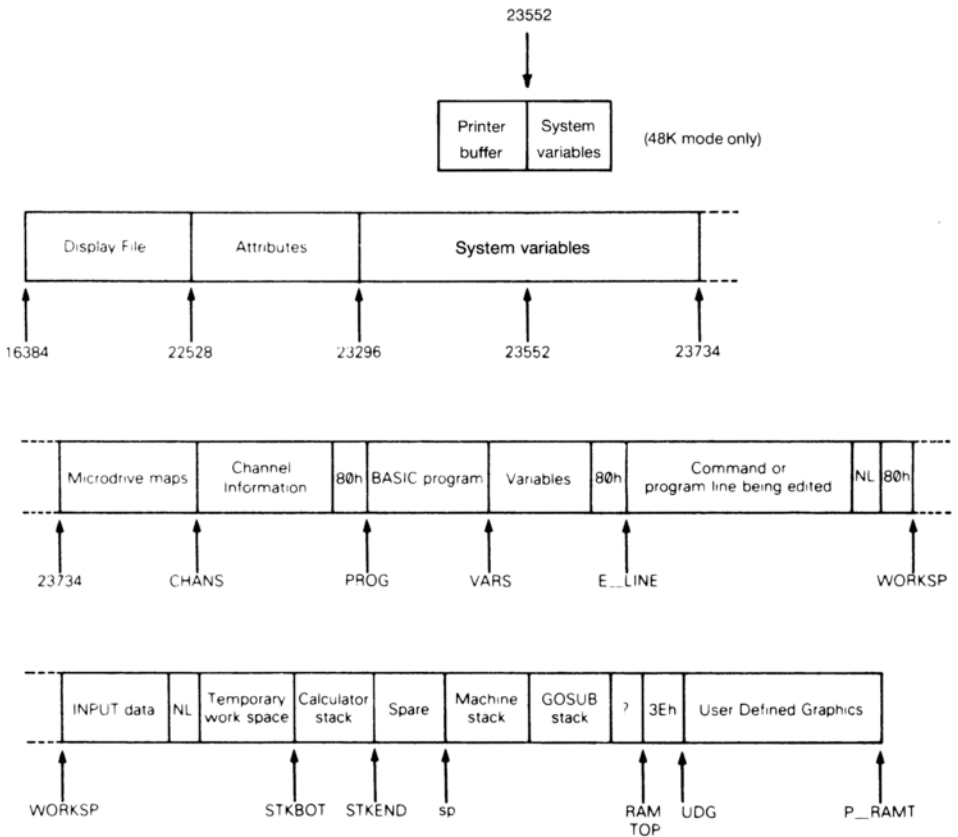
We shall now take a more detailed look at how the RAM is used. Don't bother to read this unless you're really interested.

The memory is divided into different areas (shown in the diagram ahead) for storing different kinds of information. The areas are only large enough for the information that they actually contain, and if you insert some more at a given point (for instance by adding a program line or variable), then space is made by shifting up everything above that point. Conversely, if you delete information, then everything is shifted down.

The display file stores the TV picture. It is rather curiously laid out, so you probably won't want to PEEK or POKE in it. Each character position on the screen has an 8 x 8 grid of dots; each dot can be either 0 (paper) or 1 (ink), so by using binary notation we can store the pattern as 8 bytes - one for each row. However, these 8 bytes are not stored together. The corresponding columns in the 32 characters of a single line are stored together as a scan of 32 bytes, because this is what the electron beam in the TV needs as it scans from the left hand side of the screen to the other. Since the complete picture has 24 lines of 8 scans each, you might expect the total of 172 scans to be stored in order, one after the other - well, you'd be wrong! First come the top scans of lines 0 to 7, then the next scans of lines 0 to 7, and so on to the bottom scans of lines 0 to 7; then the same for lines 8 to 15; and again for lines 16 to 23. The upshot of all this is that if you're used to a computer that uses PEEK and POKE on the screen, then you'll have to start using SCREEN\$ and PRINT AT instead (or PLOT and POINT).

The attributes are the colours and so on for each character position, using the format of `ATTR`. These are stored line by line in the order you'd expect.

The way that the computer organises its affairs changes slightly between 48 BASIC and 128 BASIC mode. The area that was the printer buffer in 48 BASIC mode, is used for extra system variables in 128 BASIC mode.



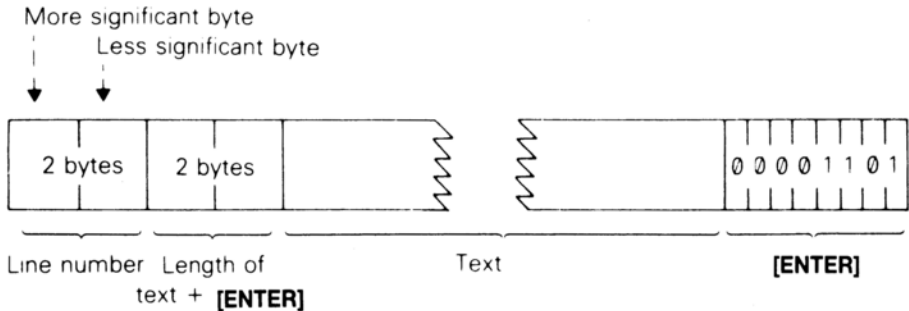
BASIC memory map

The system variables contain various pieces of information that tell the computer what sort of state it's in. They are listed fully in part 25 of this chapter, but for the moment, note that there are some (called *CHANS*, *PROG*, *VARS*, *E_LINE*, and so on) that contain the addresses of the boundaries between the various areas in memory. These are not BASIC variables, and their names will not be recognised by the +2.

The microdrive maps are only used with the microdrive. Normally there is nothing there.

The channel information contains information about the input and output devices, namely the keyboard (together with the lower half of the screen), the upper half of the screen, and the printer.

Each line of BASIC program has the form:

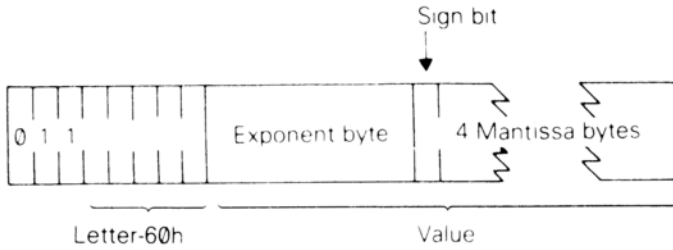


Note that, in contrast with all other cases of two-byte numbers in the Z80, the line number here is stored with its most significant byte first; that is to say, in the order that you write them down in.

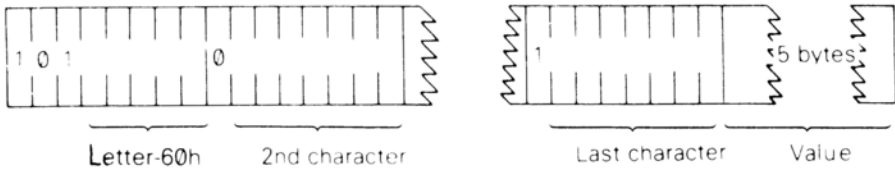
A numerical constant in the program is followed by its binary form, using the character CHR\$ 14 followed by five bytes for the number itself.

The variables have different formats according to their different natures. The letters in the names should be imagined as starting off in lower case.

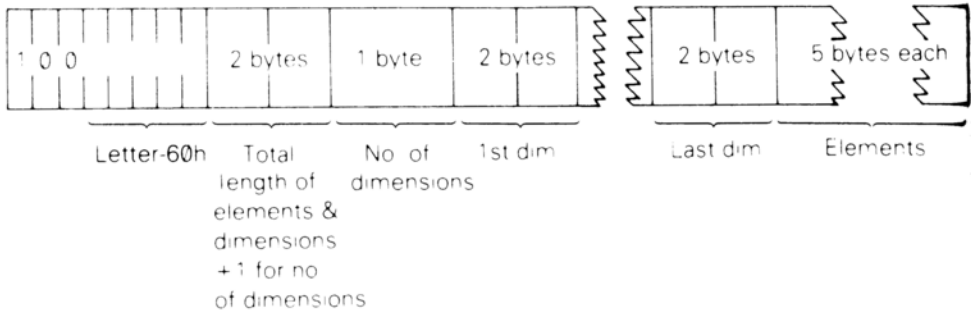
Number whose name is one letter only:



Number whose name is longer than one letter:



Array of numbers:



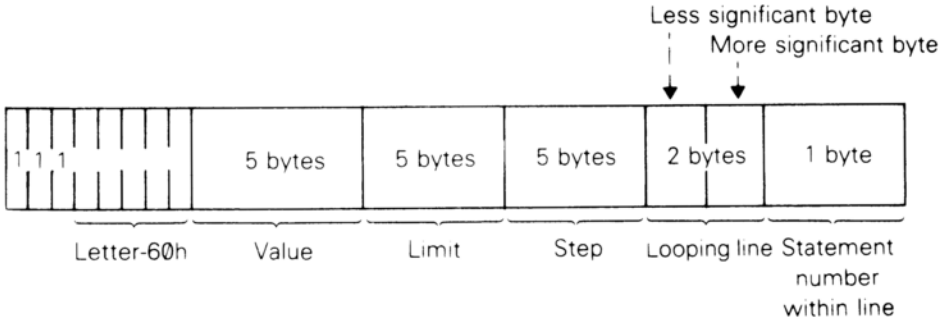
The order of the element is:

- First - the elements for which the first subscript is 1.
- Next - the elements for which the first subscript is 2.
- Next - the elements for which the first subscript is 3...
- ...and so on for all possible values of the first subscript.

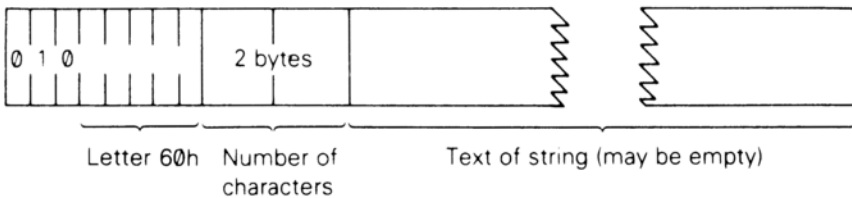
The elements with a given first subscript are ordered in the same way using the second subscript, and so on down to the last.

As an example, the elements of the 3*6 array c in part 12 of this chapter are stored in the order c(1,1) c(1,2) c(1,3) c(1,4) c(1,5) c(1,6) and c(2,1) c(2,2)... c(2,6) and c(3,1) c(3,2)... c(3,6).

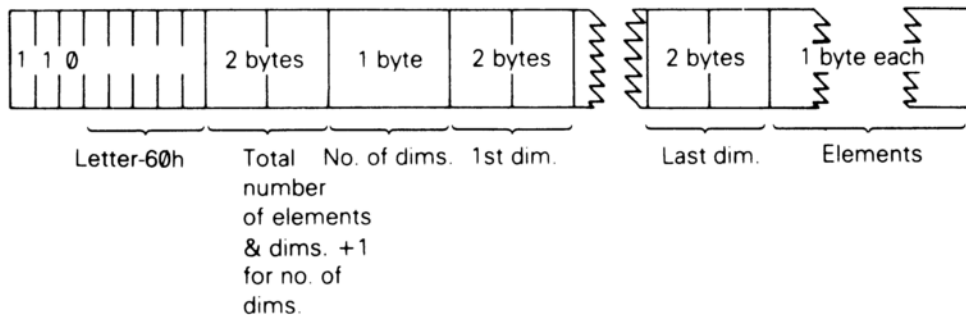
Control variable of a FOR...NEXT loop:



String:



Array of characters:



The calculator is the part of the BASIC system that deals with arithmetic, and the numbers on which it is operating are held mostly in the calculator stack.

The spare part contains the space so far unused.

The machine stack is the stack used by the Z80 processor to hold return addresses and so on.

The GO SUB stack was mentioned in part 5 of this chapter.

The byte 'pointed to' by RAMTOP has the highest address used by the BASIC system. Even NEW, which clears the RAM out, only does so as far as this - so it *doesn't* change the user-defined graphics. You can change the address RAMTOP by putting a number in a CLEAR statement, ie...

```
CLEAR new RAMTOP
```

...which does the following:

1. Clears out all the variables.
2. Clears the display file (like CLS does).
3. Resets the PLOT position to the bottom left-hand corner.
4. RESTOREs the DATA pointer.
5. Clears the GO SUB stack and puts it at the new RAMTOP (assuming that this lies between the calculator and the physical end of RAM; otherwise it leaves RAMTOP where it was).

RUN also performs a CLEAR, although it never changes RAMTOP.

Using CLEAR in this way, you can either move RAMTOP up to make more room for the BASIC by overwriting the user-defined graphics, or you can move it down to make more RAM that is preserved from NEW.

Type NEW, then CLEAR 23825 to get some idea of what happens to the machine when it fills up.

If you then try to make the +2 compute, (type in, for example PRINT 1+1) you will see the report '4 Out of memory' displayed. This means the computer has no more room for information. If you come up against this message while entering a large program, you will have to empty the memory slightly (delete a line or so) in order to control the computer.

Memory management

We mentioned earlier that there is rather more memory in the computer than the processor can comfortably deal with. While the processor can indeed only address 64K of memory at once, the extra memory can be slotted in and out of that 64K at will. Consider a TV set. Although it (and you) can only deal with one channel at a time, there are another three always there which can be selected with the right buttons. So, even though there's four times as much information as you can use at any one time, you can pick and choose which part is relevant.

It is much the same for the processor. By setting the right bits in an I/O port it can pick and choose which chunks of the 160K of memory it wants to use. For the majority of the time in BASIC it ignores most of the memory, but for games playing, having three times as much RAM is really rather useful. Look again at the +2's memory map (shown at the beginning of this section). RAMs 2 and 5 are always in the positions shown, although there's no reason why they shouldn't be in the banked section (C000h to FFFFh) -however, it would be difficult to see any use for this. The RAM banks are of two types; RAM 4 to 7 which are contended (which means they share time with the video circuitry) and RAM 0 to 3 which are uncontended (where the processor has exclusive use). Any machine code which has critical timing loops (such as music or communications programs) should keep all such routines in the uncontended banks.

The hardware switch is at I/O address 7FFDh (32765 decimal). The bit field for this address is as follows:

D0-D2 - RAM select
D3 - Screen select
D4 - ROM select
D5 - 48K lock

D2-D0 make a three bit number that selects which RAM goes into the C000h to FFFFh slot. In BASIC, RAM 0 is normally in situ, and when editing, RAM 7 is used for various buffers and 'scratchpads'. D3 switches screens; screen 0 is held in RAM 5 (beginning at 4000h) and is the one that BASIC uses, screen 1 is held in RAM 7 (beginning at C000h) and can only be used by machine code programs. It is entirely feasible to set up a screen in RAM 7 and then page it out; this leaves the entire 48K free for data and program. D4 determines whether ROM 0 (the editor ROM) or ROM 1 (the BASIC ROM) is paged into 0000h to 3FFFh. D5 is a safety feature; once this bit has been set, the machine assumes a standard 48K Spectrum configuration and all the memory paging circuitry is locked out. It cannot be turned back into a 128K machine other than by switching off or pressing the **RESET** button; however, the sound chip can still be driven by OUT.

Part 25

The system variables

Subjects covered...

POKE, PEEK

The bytes in memory from 23296 to 23733 are set aside for specific uses by the system. There are a few routines (used to keep the paging in order), and some locations called *system variables*. You can peek these to find out various things about the system, and some of them can be usefully poked. They are listed here with their uses.

There is quite a difference, as you might expect, between the system variables' area in 48 BASIC mode and 128 BASIC mode. In 48 BASIC mode, all the variables and routines below 23552 do not exist; instead there is a buffer between 23296 and 23552 which is used for controlling the printer. This was quite a popular location for small machine code routines on the 48K Spectrum, and if any of these routines are tried in 128 BASIC mode, the computer will invariably crash. Any old program that uses PEEK, POKE andUSR is therefore a safer bet if it is run in 48 BASIC mode (although it can be entered in 128 BASIC mode and transferred using the SPECTRUM command).

System variables have names, but do not confuse them with the words and names used in BASIC. The computer will not recognise the names as referring to system variables; they are given solely as mnemonics for we humans.

The abbreviations in column 1 of the table ahead have the following meanings:

X - The variables should not be poked because the system might crash.

N - Poking the variables will have no lasting effect.

R - Routine entry point. Not a variable.

The number in column 1 is the number of bytes in the variable or routine. For two bytes, the first one is the least significant byte - the reverse of what you might expect. So, to poke a value v into a two-byte variable at address n , use...

```
POKE n,v-256*INT (v/256)
POKE n+1,INT (v/256)
```

...and to peek its value, use the expression...

```
PEEK n+256*PEEK (n+1)
```

NOTES	ADDRESS	NAME	CONTENTS
R20	23296	SWAP	Paging subroutine.
R9	23316	YOUNGER	Paging subroutine.
R18	23325	ONERR	Paging subroutine.
R5	23343	PIN	RS232 input preroutine.
R22	23348	POUT	RS232 token output preroutine. This can be patched to bypass the control code filter.
R14	23370	POUT2	RS232 character output preroutine.
N2	23384	TARGET	Subroutine address in ROM 1.
X2	23386	RETADDR	Return address in ROM 0.
X1	23388	BANKM	Copy of last byte output to bank.
X1	23389	RAMRST	RST 8 instruction.
N1	23390	RAMERR	Error number, ROM 1.
2	23391	BAUD	RS232 bit period in T states/26.
N2	23393	SERFL	Second-character-received-flag, and data.
N1	23395	COL	Current column from 1 to width.
1	23396	WIDTH	Paper column width.
1	23397	TVPARS	No. of inline parameters expected by RS232.
1	23398	FLAGS3	Various flags.
N10	23399	N STR1	File name.
1	23409	HD 00	Type of file code.
2	23410	HD 0B	Length of block.
2	23412	HD 0D	Start of block.
2	23414	HD 0F	Program length.
2	23416	HD 11	Line number.
1	23418	SC 00	Second set - file type code.
2	23419	SC 08	Second set - length of block.
2	23421	SC 0D	Second set - start of block.
2	23423	SC 0F	Second set - program length.
X2	23425	OLDSP	Old SP when TSTACK in use.
X2	23427	SFNEXT	Pointer to first empty directory entry.
X3	23429	SFSPACE	Number of bytes left (17 bit).
N1	23432	ROW01	Keypad flags and row 1 image.
N1	23433	ROW23	Keypad rows 2 and 3 images.
N1	23434	ROW45	Keypad rows 4 and 5 images.
X2	23435	SYNRET	Return address for ONERR.
5	23437	LASTV	Last value printed by calculator.
2	23442	RNLINE	Current line being renumbered.
2	23444	RNFIRST	Starting line number for RNUMBER.
2	23446	RNSTEP	Incremental value for RNUMBER.
N8	23448	STRIP1	Stripe one bitmap.
N8	23456	STRIP2	Stripe two bitmap.
X	23551	TSTACK	Temporary stack grows down from here.
N8	23552	KSTATE	Used in reading the keyboard.
N1	23560	LAST K	Stores newly pressed key.

1	23561	REPDEL	Time (in 50ths of a second - in 60ths of a second in USA) that a key must be held down before it repeats. This starts off at 35, but you can P O K E in other values.
1	23562	REPPER	Delay (in 50ths of a second - in 60ths of a second in USA) between successive repeats of a key held down -initially 5.
N2	23563	DEFADD	Address of arguments of user defined function if one is being evaluated; otherwise 0.
N1	23565	KDATA	Stores 2nd byte of colour controls entered from keyboard.
N2	23566	TVDATA	Stores bytes of colour, A T and T A B controls going to TV.
X38	23568	STRMS	Addresses of channels attached to streams.
2	23606	CHARS	256 less than address of character set (which starts with space and carries on to the copyright symbol) Normally in ROM, but you can set up your own in RAM and make CHARS point to it.
1	23608	RASP	Length of warning buzz.
1	23609	PIP	Length of keyboard click.
1	23610	ERR NR	1 less than the report code. Starts off at 255 (for -1) so P E E K 2 3 6 1 0 gives 255.
X1	23611	FLAGS	Various flags to control the BASIC system.
X1	23612	TV FLAG	Flags associated with the TV.
X2	23613	ERR SP	Address of item on machine stack to be used as error return.
N2	23615	LIST SP	Address of return address from automatic listing.
N1	23617	MODE	Specifies K, L, C, E or G cursor.
2	23618	NEWPPC	Line to be jumped to.
1	23620	NSPPC	Statement number in line to be jumped to poking first NEWPPC and then NSPPC forces a jump to a specified statement in a line.
2	23621	PPC	Line number of statement currently being executed.
1	23623	SUBPPC	Number within line of statement being executed.
1	23624	BORDCR	Border colour multiplied by 8; also contains the attributes normally used for the lower half of the screen.
2	23625	E PPC	Number of current line (with program cursor).
X2	23627	VARS	Address of variables.
N2	23629	DEST	Address of variable in assignment.
X2	23631	CHANS	Address of channel data.
X2	23633	CURCHL	Address of information currently being used for input and output.
X2	23635	PROG	Address of BASIC program.
X2	23637	NXTLIN	Address of next line in program.
X2	23639	DATADD	Address of terminator of last D A T A item.
X2	23641	E LINE	Address of command being typed in.
2	23643	K CUR	Address of cursor.

X2	23645	CH ADD	Address of the next character to be interpreted - the character after the argument of PEEK, or the NEWLINE at the end of a POKE statement.
2	23647	X PTR	Address of the character after the ? marker.
X2	23649	WORKSP	Address of temporary work space.
X2	23651	STKBOT	Address of bottom of calculator stack.
X2	23653	STKEND	Address of start of spare space.
N1	23655	BREG	Calculator's b register.
N2	23656	MEM	Address of area used for calculator's memory. (Usually MEMBOT, but not always.)
1	23658	FLAGS2	More flags.
X1	23659	DF SZ	The number of lines (including one blank line) in the lower part of the screen.
2	23660	S TOP	The number of the top program line in automatic listings.
2	23662	OLDPPC	Line number to which CONTINUE jumps.
1	23664		Number within line of statement to which CONTINUE jumps.
N1	23665	OSPCC	
		FLAGX	Various flags.
N2	23666	STRLEN	Length of string type destination in assignment.
N2	23668	T ADDR	Address of next item in syntax table (very unlikely to be useful).
2	23670	SEED	The seed for RND. This is the variable that is set by RANDOMIZE.
3	23672	FRAMES	3 byte (least significant byte first), frame counter incremented every 20mS. (See part 18 of this chapter.)
2	23675	UDG	Address of 1st user-defined graphic. You can change this, for instance, to save space by having fewer user-defined graphics.
1	23677	COORDS	x-coordinate of last point plotted.
1	23678		y-coordinate of last point plotted.
1	23679	P POSN	33-column number of printer position.
1	23680	PR CC	Least significant byte of address of next position for LPRINT to print at (in printer buffer).
1	23681		Not used.
2	23682	ECHO E	33-column number and 24-line number (in lower half) of end of input buffer.
2	23684	DF CC	Address in display file of PRINT position.
2	23686	DF CCL	Like DF CC for lower part of screen.
X1	23688	S POSN	33-column number for PRINT position.
X1	23689		24-line number for PRINT position.
X2	23690	SPOSNL	Like S POSN for lower part.
1	23692	SCR CT	Counts scrolls - it is always 1 more than the number of scrolls that will be done before stopping with 'scroll?' If you keep poking this with a number bigger than 1 (say 255), the screen will scroll on and on without asking you.

1	23693	ATTRP	Permanent current colours, etc., (as set up by colour statements).
1	23694	MASK P	Used for transparent colours etc. Any bit that is 1 shows that the corresponding attribute bit is taken not from ATTRP, but from what is already on the screen.
N1	23695	ATTRT	Temporary current colours, etc., (as set up by colour items).
N1	23696	MASK T	Like MASK P, but temporary.
1	23697	PFLAG	More flags.
N30	23698	MEMBOT	Calculator's memory area - used to store numbers that cannot conveniently be put on the calculator stack.
2	23728		Not used.
2	23730	RAMTOP	Address of last byte of BASIC system area.
2	23732	P-RAMT	Address of last byte of physical RAM.

Exercise...

1. This program shows you 22 bytes of the variables area (from *KSCAN* onwards)...

```

10 FOR n=0 TO 21
20 PRINT PEEK (PEEK 23627+256*PEEK 23628+n)
30 NEXT N

```

Try to match up the control variable *n* with the descriptions above. Now change line 20 to...

```

20 PRINT PEEK (23755+n)

```

This shows you the first 22 bytes of the program area. Match these up with the program itself.

Part 26

Using machine code

Subjects covered...

U S R with numeric argument

This section is written for those who understand Z80 *machine code*, ie. the set of instructions that the Z80 processor chip uses. If you do not, but would like to, there are plenty of books about it. You should get one called something along the lines of... 'Z80 machine code (or assembly language) for the absolute beginner', and if it mentions the '+2' or other computers in the ZX Spectrum range, so much the better.

Machine code programs are normally written in *assembly language*, which, although cryptic, is not too difficult to understand with practice. You can see the assembly language instructions in part 27 of this chapter. However, to run them on the +2 you need to code the program into a sequence of bytes - in this form it is called machine code. This translation is usually done by the computer itself, using a program called an assembler. There is no assembler built in to the +2, but you will be able to buy one on cassette. Failing that, you will have to do the translation yourself, provided that the program is not too long.

Let's take as an example the program...

```
ld bc, 99
ret
```

...which loads the *bc* register pair with 99. This translates into the four machine code bytes 1, 99, 0 (for *ld bc, 99*) and 201 (for *ret*). (If you look up codes 1 and 201 in part 27 ahead, you will find that 1 corresponds to *ld bc, NN* - where *NN* stands for any two-byte number; and 201 corresponds to *ret*.)

When you have got your machine code program, the next step is to get it into the computer - (an assembler would probably do this automatically). You need to decide whereabouts in memory to locate it - the best thing is to make extra space for it between the BASIC area and the user-defined graphics.

If you type...

```
CLEAR 65267
```

...this will give you a space of 100 (for good measure) bytes starting at address 65268.

To put in the machine code program, you would run a BASIC program something like...

```
10 LET a=65268
20 READ n: POKE a,n
30 LET a=a+1: GO TO 20
40 DATA 1,99,0,201
```

(This will stop with the report 'E O u t o f DATA' when it has filled in the four bytes you specified.)

To run the machine code, you use the function USR - but this time with a numeric argument, ie. the starting address. Its result is the value of the *bc* register on return from the machine code program, so if you type...

```
PRINT USR 65268
```

...you will get the answer 99.

The return address to BASIC is 'stacked' in the usual way, so return is by a Z80 *ret* instruction. You should not use the *iy* and *i* registers in a machine code routine that expects to use the BASIC interrupt mechanism. You should also not load *i* with values between 40h and 7Fh (even if you never use *IM 2*). Values between C0h and FFh for *i* should also be avoided if contended memory (ie. RAM 4 to 7) is to be paged in between C000h and FFFFh. This is due to an interaction between the video controller and the Z80 refresh mechanism, and can cause otherwise inexplicable crashes, screen corruption or other undesirable effects. Thus, you should only vector *IM 2* interrupts to between 8000h and BFFFh, unless you are very confident of your memory mapping.

There are a number of standard pitfalls when programming a *banked* system such as the *+2* from machine code. If you are experiencing problems, check that your stack is not being paged out during interrupts, and that your interrupt routine is always where you expect it to be! (it is advisable to disable interrupts during paging operations). It is also recommended that you keep a copy of the current bank register setting in unpaged RAM somewhere, as the port is write-only. BASIC and the editor use the system variable *BANK M*.

You can save your machine code program easily enough with...

```
SAVE "some name" CODE 65268,4
```

On the face of it, there is no way of saving the program so that when loaded it automatically runs itself; however, you can get round this by using the short BASIC program...

```
10 LOAD "" CODE 65268,4
20 PRINT USR 65268
```

...which must be saved to cassette *just before* the machine code, using the command (for example)...

```
SAVE "loader" LINE 0
```

...then you may save the machine code using (for example)...

```
SAVE "m code" CODE 65268,4
```

...after which, you may run the machine code from BASIC using the single command...

```
LOAD "loader"
```

...which loads and automatically runs the BASIC program which in turn loads and runs the machine code.

Part 27

Spectrum character set

Subjects covered...

Control codes
 Characters
 Z80 assembler mnemonics

This is the complete Spectrum character set, with codes in decimal and hex. If one imagines the codes as being Z80 machine code instructions, then the right hand columns give the corresponding assembly language mnemonics. As you may be aware, certain Z80 instructions are 'compounds' starting with CBh or EDh; these are shown in the two right hand columns. Where a character changes (between 48K and 128K modes), the 48K version is given in brackets after the 128K one.

CODE	CHARACTER	HEX	Z80 ASSEMBLER	-AFTER CBh	-AFTER EDh
0] not used	00	nop	cb	
1		01	ld bc,NN	rlc c	
2		02	ld (bc),a	rlc d	
3		03	inc bc	rlc e	
4		04	inc b	rlc h	
5		05	dec b	rlcl	
6	PRINT comma	06	ld b,N	rlc (hl)	
7	[EDIT]	07	rlca	rlca	
8	cursor left ↵	08	ex af,af	rrc b	
9	cursor right ⇨	09	add hl,bc	rrc c	
10	cursor down ⇩	0A	ld a,(bc)	rrc d	
11	cursor up ⇧	0B	dec bc	rrc e	
12	[DELETE]	0C	inc c	rrc h	
13	[ENTER]	0D	dec c	rrcl	
14	number	0E	ld c,N	rrc (hl)	
15	not used	0F	rrca	rrc a	
16	INK control	10	djnz DIS	rl b	
17	PAPER control	11	ld de,NN	rl c	
18	FLASH control	12	ld (de),a	rl d	
19	BRIGHT control	13	inc de	rl e	
20	INVERSE control	14	inc d	rl h	
21	OVER control	15	dec d	rl l	
22	AT control	16	ld d,N	rl (hl)	
23	TAB control	17	rla	rl a	
24] not used	18	jr DIS	rr b	
25		19	add hl,de	rr c	

26	}	not used	1A	ld a,(de)	rr d	
27			1B	dec de	rr e	
28			1C	inc e	rr h	
29			1D	dec e	rr l	
30			1E	ld e,N	rr (hl)	
31			1F	rr a	rr a	
32	space		20	jr nz,DIS	sla b	
33	!		21	ld hl,NN	sla c	
34	"		22	ld (NN),hl	sla d	
35	#		23	inc hl	sla e	
36	\$		24	inc h	sla h	
37	%		25	dec h	sla l	
38	&		26	ld h,N	sla (hl)	
39	'		27	daa	sla a	
40	(28	jr z,DIS	sra b	
41)		29	add hl,hl	sra c	
42	*		2A	ld hl,(NN)	sra d	
43	+		2B	dec hl	sra e	
44	,		2C	inc l	sra h	
45	-		2D	dec l	sra l	
46	.		2E	ld l,N	sra (hl)	
47	/		2F	cpl	sra a	
48	Ø		30	jr nc,DIS		
49	1		31	ld sp,NN		
50	2		32	ld (NN),a		
51	3		33	inc sp		
52	4		34	inc (hl)		
53	5		35	dec (hl)		
54	6		36	ld (hl),N		
55	7		37	scf		
56	8		38	jr c,DIS	srl b	
57	9		39	add hl,sp	srl c	
58	:		3A	ld a,(NN)	srl d	
59	;		3B	dec sp	srl e	
60	<		3C	inc a	srl h	
61	=		3D	dec a	srl l	
62	>		3E	ld a,N	srl (hl)	
63	?		3F	ccf	srl a	
64	@		40	ld b,b	bit 0,b	in b,(c)
65	A		41	ld b,c	bit 0,c	out (c),b
66	B		42	ld b,d	bit 0,d	sbc hl,bc
67	C		43	ld b,e	bit 0,e	ld (NN),bc
68	D		44	ld b,h	bit 0,h	neg
69	E		45	ld b,l	bit 0,l	retn
70	F		46	ld b,(hl)	bit 0,(hl)	im 0
71	G		47	ld b,a	bit 0,a	ld i,a
72	H		48	ld c,b	bit 1,b	in c,(c)

73	I	49	ld c,c	bit 1,c	out (c),c
74	J	4A	ld c,d	bit 1,d	adc hl,bc
75	K	4B	ld c,e	bit 1,e	ld bc,(NN)
76	L	4C	ld c,h	bit 1,h	
77	M	4D	ld c,l	bit 1,l	reti
78	N	4E	ld c,(hl)	bit 1,(hl)	
79	O	4F	ld c,a	bit 1,a	ld r,a
80	P	50	ld d,b	bit 2,b	in d,(c)
81	Q	51	ld d,c	bit 2,c	out (c),d
82	R	52	ld d,d	bit 2,d	sbc hl,de
83	S	53	ld d,e	bit 2,e	ld (NN),de
84	T	54	ld d,h	bit 2,h	
85	U	55	ld d,l	bit 2,l	
86	V	56	ld d,(hl)	bit 2,(hl)	im 1
87	W	57	ld d,a	bit 2,a	ld a,i
88	X	58	ld e,b	bit 3,b	in e,(c)
89	Y	59	ld e,c	bit 3,c	out (c),e
90	Z	5A	ld e,d	bit 3,d	adc hl,de
91	[5B	ld e,e	bit 3,e	ld de,(NN)
92	\	5C	ld e,h	bit 3,h	
93]	5D	ld e,l	bit 3,l	
94	↑	5E	ld e,(hl)	bit 3,(hl)	im 2
95	—	5F	ld e,a	bit 3,a	ld a,r
96	£	60	ld h,b	bit 4,b	in h,(c)
97	a	61	ld h,c	bit 4,c	out (c),h
98	b	62	ld h,d	bit 4,d	sbc hl,hl
99	c	63	ld h,e	bit 4,e	ld (NN),hl
100	d	64	ld h,h	bit 4,h	
101	e	65	ld h,l	bit 4,l	
102	f	66	ld h,(hl)	bit 4,(hl)	
103	g	67	ld h,a	bit 4,a	rrd
104	h	68	ld l,b	bit 5,b	in l,(c)
105	i	69	ld l,c	bit 5,c	out (C),l
106	j	6A	ld l,d	bit 5,d	adc hl,hl
107	k	6B	ld l,e	bit 5,e	ld hl,(NN)
108	l	6C	ld l,h	bit 5,h	
109	m	6D	ld l,l	bit 5,l	
110	n	6E	ld l,(hl)	bit 5,(hl)	
111	o	6F	ld l,a	bit 5,a	rld
112	p	70	ld (hl),b	bit 6,b	in f,(c)
113	q	71	ld (hl),c	bit 6,c	
114	r	72	ld (hl),d	bit 6,d	sbc hl,sp
115	s	73	ld (hl),e	bit 6,e	ld (NN),sp
116	t	74	ld (hl),h	bit 6,h	
117	u	75	ld (hl),l	bit 6,l	
118	v	76	halt	bit 6,(hl)	
119	w	77	ld (hl),a	bit 6,a	

120	x		78	ld a,b	bit 7,b	in a,(c)
121	y		79	ld a,c	bit 7,c	out (c),a
122	z		7A	ld a,d	bit 7,d	adc hl,sp
123	{		7B	ld a,e	bit 7,e	ld sp,(NN)
124			7C	ld a,h	bit 7,h	
125	}		7D	ld a,l	bit 7,l	
126	~		7E	ld a,(hl)	bit 7,(hl)	
127	©		7F	ld a,a	bit 7,a	
128	□		80	add a,b	res 0,b	
129	■		81	add a,c	res 0,c	
130	▣		82	add a,d	res 0,d	
131	▤		83	add a,e	res 0,e	
132	▥		84	add a,h	res 0,h	
133	▦		85	add a,l	res 0,l	
134	▧		86	add a,(hl)	res 0,(hl)	
135	▨		87	add a,a	res 0,a	
136	▩		88	adc a,b	res 1,b	
137	▪		89	adc a,c	res 1,c	
138	▫		8A	adc a,d	res 1,d	
139	▬		8B	adc a,e	res 1,e	
140	▮		8C	adc a,h	res 1,h	
141	▯		8D	adc a,l	res 1,l	
142	▰		8E	adc a,(hl)	res 1,(hl)	
143	▱		8F	adc a,a	res 1,a	
144	(a)	user graphics	90	sub b	res 2,b	
145	(b)		91	sub c	res 2,c	
146	(c)		92	sub d	res 2,d	
147	(d)		93	sub e	res 2,e	
148	(e)		94	sub h	res 2,h	
149	(f)		95	sub l	res 2,l	
150	(g)		96	sub (hl)	res 2,(hl)	
151	(h)		97	sub a	res 2,a	
152	(i)		98	sbc a,b	res 3,b	
153	(j)		99	sbc a,c	res 3,c	
154	(k)		9A	sbc a,d	res 3,d	
155	(l)		9B	sbc a,e	res 3,e	
156	(m)		9C	sbc a,h	res 3,h	
157	(n)	9D	sbc a,l	res 3,l		
158	(o)	9E	sbc a,(hl)	res 3,(hl)		
159	(p)	9F	sbc a,a	res 3,a		
160	(q)	A0	and b	res 4,b	ldi	
161	(r)	A1	and c	res 4,c	cpi	
162	(s)	A2	and d	res 4,d	ini	
163	SPECTRUM (t)	A3	and e	res 4,e	outi	
164	PLAY (u)]	A4	and h	res 4,h		
165	RND	A5	and l	res 4,l		
166	INKEY\$	A6	and (hl)	res 4,(hl)		

167	PI	A7	and a	res 4,a	
168	FN	A8	xor b	res 5,b	ldd
169	POINT	A9	xor c	res 5,c	cpd
170	SCREEN\$	AA	xor d	res 5,d	ind
171	ATTR	AB	xor e	res 5,e	outd
172	AT	AC	xor h	res 5,h	
173	TAB	AD	xor l	res 5,l	
174	VAL\$	AE	xor (hl)	res 5,(hl)	
175	CODE	AF	xor a	res 5,a	
176	VAL	B0	or b	res 6,b	ldir
177	LEN	B1	or c	res 6,c	cpir
178	SIN	B2	or d	res 6,d	inir
179	COS	B3	or e	res 6,e	otir
180	TAN	B4	or h	res 6,h	
181	ASN	B5	or l	res 6,l	
182	ACS	B6	or (hl)	res 6,(hl)	
183	ATN	B7	or a	res 6,a	
184	LN	B8	cp b	res 7,b	laddr
185	EXP	B9	cp c	res 7,c	cpdr
186	INT	BA	cp d	res 7,d	indr
187	SQR	BB	cp e	res 7,e	otdr
188	SGN	BC	cp h	res 7,h	
189	ABS	BD	cp l	res 7,l	
190	PEEK	BE	cp (hl)	res 7,(hl)	
191	IN	BF	cp a	res 7,a	
192	USR	C0	ret nz	set 0,b	
193	STR\$	C1	pop bc	set 0,c	
194	CHR\$	C2	jp nz,NN	set 0,d	
195	NOT	C3	jp NN	set 0,e	
196	BIN	C4	call nz,NN	set 0,h	
197	OR	C5	push bc	set 0,l	
198	AND	C6	add a,N	set 0,(hl)	
199	<=	C7	rst 0	set 0,a	
200	>=	C8	ret z	set 1,b	
201	<>	C9	ret	set 1,c	
202	LINE	CA	jp z,NN	set 1,d	
203	THEN	CB		set 1,e	
204	TO	CC	call z,NN	set 1,h	
205	STEP	CD	call NN	set 1,l	
206	DEF FN	CE	adc a,N	set 1,(hl)	
207	CAT	CF	rst 8	set 1,a	
208	FORMAT	D0	pop de	set 2,b	
209	MOVE	D1	pop de	set 2,c	
210	ERASE	D2	jp nc,NN	set 2,d	
211	OPEN #	D3	out (N),a	set 2,e	
212	CLOSE #	D4	call nc,NN	set 2,h	
213	MERGE	D5	push de	set 2,l	

214	VERIFY	D6	sub N	set 2,(hl)
215	BEEP	D7	rst 16	set 2,a
216	CIRCLE	D8	ret c	set 3,b
217	INK	D9	exx	set 3,c
218	PAPER	DA	jp c,NN	set 3,d
219	FLASH	DB	in a,(N)	set 3,e
220	BRIGHT	DC	call c,NN	set 3,h
221	INVERSE	DD	prefixes instructions using ix	set 3,l
222	OVER	DE	sbc a,N	set 3,(hl)
223	OUT	DF	rst 24	set 3,a
224	LPRINT	E0	ret po	set 4,b
225	LLIST	E1	pop hl	set 4,c
226	STOP	E2	jp po,NN	set 4,d
227	READ	E3	ex (sp),hl	set 4,e
228	DATA	E4	call po,NN	set 4,h
229	RESTORE	E5	push hl	set 4,l
230	NEW	E6	and N	set 4,(hl)
231	BORDER	E7	rst 32	set 4,a
232	CONTINUE	E8	ret pe	set 5,b
233	DIM	E9	jp (hl)	set 5,c
234	REM	EA	jp pe,NN	set 5,d
235	FOR	EB	ex de,hl	set 5,e
236	GOTO	EC	call pe,NN	set 5,h
237	GO SUB	ED		set 5,l
238	INPUT	EE	xor N	set 5,(hl)
239	LOAD	EF	rst 40	set 5,a
240	LIST	F0	ret p	set 6,b
241	LET	F1	pop af	set 6,c
242	PAUSE	F2	jp p,NN	set 6,d
243	NEXT	F3	di	set 6,e
244	POKE	F4	call p,NN	set 6,h
245	PRINT	F5	push af	set 6,l
246	PLOT	F6	or N	set 6,(hl)
247	RUN	F7	rst 48	set 6,a
248	SAVE	F8	ret m	set 7,b
249	RANDOMIZE	F9	ld sp,hl	set 7,c
250	IF	FA	jp m,NN	set 7,d
251	CLS	FB	ei	set 7,e
252	DRAW	FC	call m,NN	set 7,h
253	CLEAR	FD	prefixes instructions using iy	set 7,l
254	RETURN	FE	cp N	set 7,(hl)
255	COPY	FF	rst 56	set 7,a

Part 28

Reports

Subjects covered...

Screen display messages
Error messages
Reports
CONTINUE

Reports appear at the bottom of the screen whenever the **+2** has stopped executing some BASIC. They explain why it has stopped - be it for some natural reason, or because an error has occurred.

The report has a code number or letter (so that you can refer to the table here), a brief message explaining what happened, and the line number (and the statement number within the line) where it stopped. (A command is shown as line 0. Within a line, statement 1 is at the beginning, statement 2 comes after the first colon (or THEN), and so on.)

The behaviour of CONTINUE depends very much on the reports. Normally, CONTINUE goes to the line and statement specified in the last report, but there are exceptions with reports 0, 9 and D.

Here is a table showing all the reports. It also tells you in what circumstances the report can occur, and this refers you to part 30 of this chapter. For instance, the error 'A Invalid argument' can occur with SQR, IN, ACS and ASN and the entries for these in part 30 tell you exactly which arguments are invalid.

CODE	MEANING	SITUATION
0	OK Successful completion, or jump to a line number bigger than any existing. This report does not change the line and statement jumped to by CONTINUE.	Any
1	NEXT without FOR The control variable does not exist (it has not been set up by a FOR statement), but there is an ordinary variable with the same name.	NEXT
2	Variable not found For a simple variable this will happen if the variable is used before it has been assigned to by a LET, READ or INPUT statement, or loaded from cassette, or set up in a FOR statement. For a subscripted variable it will happen if the variable is used before it has been dimensioned in a DIM statement, or loaded from cassette.	Any

3	Subscript wrong A subscript is beyond the dimension of the array, or there are the wrong number of subscripts. If the subscript is negative or bigger than 65535, then error B will result.	Subscripted variables, Substrings
4	Out of memory There is not enough room in the computer for what you are trying to do. If the computer really seems to be stuck in this state, you may have to clear out the command line using [DELETE] and then delete a program line or two (with the intention of putting them back afterwards) to give yourself room to manoeuvre.	LET, INPUT, FOR, DIM, GO SUB, LOAD MERGE. Sometimes during expression evaluation
5	Out of screen An INPUT statement has tried to generate more than 23 lines in the lower half of the screen. Also occurs with PRINT AT 22,xx.	INPUT, PRINT AT
6	Number too big Calculations have led to a number greater than approximately 10^{38} .	Any arithmetic
7	RETURN without GO SUB There has been one more RETURN than there were GO SUBs.	RETURN
8	End of File	Microdrive, etc. operations
9	STOP statement After this, CONTINUE will not repeat the STOP, but carries on with the statement after.	STOP
A	Invalid argument The argument for a function is unsuitable (for some reason).	SQN, LN, ASN, ACS, USR (with string argument)
B	Integer out of range When an integer is required, the floating point argument is rounded to the nearest integer. If this is outside a suitable range, then error B results.	RUN, RANDOMIZE, POKE, DIM, GO TO, GO SUB, LIST, LLIST, PAUSE, PLOT, CHR\$, PEEK, USR (with numeric argument)
	For array access, see also Error 3.	Array access

C	<p>Nonsense in BASIC The text of the (string) argument does not form a valid expression. Also used when the argument for a function or command is outrageously wrong.</p>	VAL, VAL\$
D	<p>BREAK - CONT repeats [BREAK] was pressed during some peripheral operation. The behaviour of CONTINUE after this report is normal in that it repeats the statement. Compare with report L.</p>	LOAD, SAVE, VERIFY, MERGE, Also used when the computer asks 'scroll?' and you press N, [BREAK] or the space bar
E	<p>Out of DATA You have tried to READ past the end of the DATA list.</p>	READ
F	<p>Invalid file name SAVE with name empty or longer than 10 characters.</p>	SAVE
G	<p>No room for line There is not enough room left in memory to accommodate the new program line.</p>	Entering a line into the program
H	<p>STOP in INPUT Some INPUT data started with STOP. Unlike the case with report 9, after report H, CONTINUE will behave normally, by repeating the INPUT statement.</p>	INPUT
I	<p>FOR without NEXT There was a FOR loop to be executed no times (eg. FOR n = 1 TO 0) and the corresponding NEXT statement could not be found.</p>	FOR
J	<p>Invalid I/O device</p>	Microdrive, etc. operations
K	<p>Invalid colour The number specified is not an appropriate value.</p>	INK, PAPER, BORDER, FLASH, BRIGHT, INVERSE, OVER; also after one of the corresponding control characters
L	<p>BREAK into program [BREAK] pressed. This is detected between two statements. The line and statement number in the report refer to the statement before [BREAK] was pressed, but CONTINUE goes to the statement after (allowing for any jumps to be done), so it does not repeat any statements.</p>	Any

M	RAMTOP_no good The number specified for RAMTOP is either too big or too small.	CLEAR; possibly in RUN
N	Statement lost Jump to a statement that no longer exists.	RETURN, NEXT, CONTINUE
O	Invalid Stream	Microdrive, etc. operations
P	FNwithout DEF User-defined function used without a corresponding DEF in the program.	FN
Q	Parameter error Wrong number of arguments, or one of them is the wrong type (string instead of number or vice versa).	FN
R	Tape loading error A file on cassette was found but for some reason could not be read in, or would not verify.	VERIFY, LOAD or MERGE
a	MERGE error MERGE ! would not execute for some reason - either size or file type wrong.	MERGE !
b	Wrong file type A file of an inappropriate type was specified during silicon disc operation, for instance a CODE file in LOAD ! "name".	MERGE !, LOAD !
c	CODE error The size of the file would lead to overrun of top of memory.	LOAD ! file CODE
d	Too many brackets Too many brackets around a repeated phrase in one of the arguments.	PLAY
e	File already exists The file name specified has already been used.	SAVE !
f	Invalid name The file name specified is empty or above 10 characters in length.	ERASE !
h	File does not exist There is no file in the silicon disc that has the name specified.	LOAD ! ERASE !

i	Invalid device The device name following the <code>FORMAT</code> command does not exist or correspond to a physical device.	FORMAT
j	Invalid baud rate The baud rate for the RS232 was set to zero.	FORMAT
k	Invalid note name <code>PLAY</code> came across a note or command it didn't recognise, or a command which was in lower case.	PLAY
l	Number too big A parameter for a command is an order of magnitude too big.	PLAY
m	Note out of range A series of sharps or flats has taken a note beyond the range of the sound chip.	PLAY
n	Out of range A parameter for a command is too big or too small. If the error is very large error <code>l</code> results.	PLAY
o	Too many tied notes An attempt was made to tie too many notes together.	PLAY

Part 29

Reference information

Subjects covered...

Hardware

The **+2** is designed around the Z80A microprocessor, which runs at a speed of 3.54 MHz (3.54 million cycles per second).

The **+2**'s memory is divided into 32K ROM and 128K RAM, arranged in 16K pages. The two ROM pages (0-1) are mapped into the bottom 16K (0-3FFFh) of the memory map. The eight RAM pages (0-7) are mapped into the top 16K (C000h-FFFFh) of the memory map. RAM page 5 is also mapped into the range 4000h-7FFFh, and RAM page 2 is mapped to range 8000h-BFFFh.

Physically speaking, the ROM is a single 32K device (similar to a 27256), which is treated by the system as two 16K chips. The RAM is composed of sixteen 64K x 1-bit chips (4164), some of which (RAM banks 4-7) are time-shared between the circuitry that produces the TV picture (more of which later) and the Z80A. The other eight (RAM banks 0-3) are for the exclusive use of the Z80A, as is the ROM.

The Uncommitted Logic Array (ULA) handles most of the I/O, like keyboard, datacorder and screen handling. It converts bytes in memory into patterns and colours on screen, and allows the Z80A to scan the keyboard and read and write data to cassette.

The three-channel sound is produced by the AY-3-8912 - a very popular sound chip, and this device also controls the **RS232/MIDI** and **KEYPAD** ports. The way in which it works is quite complex, and the putative experimenter is advised to get the AY-3-8912 data sheet. The following information should be enough to get things underway, however. The sound chip contains sixteen registers which are selected by writing first to the address write port (I/O address FFFDh - 65533 decimal) with the register number, and then reading the register value (same address) or writing to the data register write address (BFFDh - 49149 decimal). Once a register has been selected, any number of data read/writes can be done; the address write port need only be re-written if a different register needs to be accessed.

The basic clock frequency of the circuit is 1.7734 MHz (to 0.01%).

The registers do the following:

- R0 - Fine tone control channel A
- R1 - Coarse tone control channel A
- R2 - Fine tone control channel B
- R3 - Coarse tone control channel B
- R4 - Fine tone control channel C
- R5 - Coarse tone control channel C

The tone of a channel is a 12-bit value taken from the sum of D3-D0 of the coarse register, and D7-D0 of the the fine register. The basic unit of tone is the clock frequency divided by 16 (ie. 110.83 KHz), and with a 12 bit counter range, frequencies from 27Hz to 110 KHz can be generated.

R6 - Noise Generator Control, D4-D0

The period of the noise source is taken by counting down the lower 5 bits of the noise register every sound clock period divided by 16.

R7 - Mixer and I/O control

- D7 - not used
- D6 - 1 = input port, 0 = output port
- D5 - Channel C noise
- D4 - Channel B noise
- D3 - Channel A noise
- D2 - Channel C tone
- D1 - Channel B tone
- D0 - Channel A tone

This register controls both the mixing of noise and tone values for each channel, and the direction of the 8-bit I/O port. A zero in a mix bit indicates that the function is enabled.

- R8 - Amplitude control channel A
- R9 - Amplitude control channel B
- RA - Amplitude control channel C

- D4 - 1 = use envelope generator
- 0 = use value of D3-D0 for amplitude
- D3-D0 - Amplitude

These three registers control the amplitude of each channel and whether or not it is modulated by the envelope registers.

- RB - Envelope coarse period control
- RC - Envelope fine period control

The eight bit values in RB+RC are summed to produce a 16 bit number which is counted down in units of 256 times the sound clock. Envelope frequencies can be between 0.1Hz and 6KHz.

RD - Envelope control

- D3 - Continue
- D2 - Attack
- D1 - Alternate
- D0 - Hold

The diagram of envelope shapes (in part 19 of this chapter) gives a graphic illustration of the possible settings for this register.

Part 30

The BASIC

Subjects covered...

Number handling
Variables
Strings
Functions
Brief summary of keywords
Mathematical operations

Numbers are stored to an accuracy of 9 or 10 digits. The largest number you can get is about 10^{38} , and the smallest (positive) number is about 4×10^{-39} .

A number is stored in the **+2** in *floating point binary* with one exponent byte e ($1 \leq e \leq 255$), and four mantissa bytes m ($1/2 \leq m < 1$). This represents the number $m \times 2^{e-128}$.

Since $1/2 \leq m < 1$, the most significant bit of the mantissa m is always 1. Therefore, in actual fact we can replace it with a bit to show the sign - 0 for positive numbers, 1 for negative.

Small integers have a special representation in which the first byte is 0, the second is a sign byte (0 or FFh) and the third and fourth are the integer itself (in twos complement form) with the least significant byte first.

Numeric variables have names of arbitrary length, starting with a letter and continuing with letters and digits. Spaces are ignored and all letters are converted internally to lower-case letters.

Control variables of FOR...NEXT loops have names a single letter long.

Numeric arrays have names a single letter long, which may be the same as the name of a simple variable. They may have many dimensions of arbitrary size. Subscripts start at 1.

Strings are completely flexible in length. The name of a string consists of a single letter followed by \$.

String arrays can have many dimensions of arbitrary size. The name is a single letter followed by \$ and may not be the same as the name of a simple string variable. All the strings in a given array have the same fixed length, which is specified as an extra final dimension in the DIM statement. Subscripts start at 1.

Slicing: Substrings of strings may be specified using *slicers*. A slicer can be one of the following:

(i) empty

...or...

(ii) a numerical expression

...or...

(iii) *optional numerical expression* T O *optional numerical expression* and is used in expressing a substring by either:

(a) string expression (slicer)

...or...

(b) string array variable (subscript,... subscript, slicer)

...which is the same as...

string array variable (subscript... subscript) (slicer)

In (a), suppose the string expression has the value s \$, then if the slicer is empty, the result is s \$ (considered as a substring of itself).

If the slicer is a numerical expression with value m, then the result is the mth character of s \$ (a substring of length 1).

If the slicer has the form (iii), then suppose the first numerical expression has the value m (the default value is 1), and the second, n (the default value is the length of s \$). If $1 \leq m \leq n \leq \text{length of s \$}$, then the result is the substring of s \$ starting with the mth character and ending with the nth.

If $0 < n < m$, then the result is the empty string. Otherwise, error 3 results.

Slicing is performed before functions or operations are evaluated, unless brackets dictate otherwise.

Substrings can be assigned to (see L E T). If a string quote is to be written in a string literal, then it must be doubled.

Functions

The argument of a function does not need brackets if it is a constant or a variable (optionally subscripted or sliced).

FUNCTION	TYPE OF ARGUMENT	RESULT
ABS	number	Absolute magnitude.
ACS	number	Arccosine in radians. Error A if x not in the range -1 to +1.
AND	binary operation, right operand always a number numeric left operand: string left operand:	$a \text{ AND } b \begin{cases} a & \text{if } b < > 0 \\ 0 & \text{if } b = 0 \end{cases}$ $a\$ \text{ AND } b \begin{cases} a\$ & \text{if } b < > 0 \\ "" & \text{if } b = 0 \end{cases}$ <p>AND has priority 3.</p>
ASN	number	Arcsine in radians. Error A if x not in the range -1 to +1.
ATN	number	Arctangent in radians.
ATTR	two arguments, x and y, both numbers; enclosed in brackets	A number whose binary form codes the attributes of line x, column y on the screen. Bit 7 (most significant) is 1 for flashing, 0 for steady. Bit 6 is 1 for bright, 0 for normal. Bits 5 to 3 are the paper colour. Bits 2 to 0 are ink colour. Error B unless $0 \leq x \leq 23$ and $0 \leq y \leq 31$.
BIN		This is not really a function, but an alternative notation for numbers: BIN followed by a sequence of 0s and 1s is the number with such a representation in binary.
CHR\$	number	The character whose code is x, rounded to the nearest integer.

CODE	string	The code of the first character in x (or 0 if x is the empty string).
COS	number (in radians)	Cosine x.
EXP	number	e ^x .
FN		FN followed by a letter calls up a user-defined function (see DEF). The arguments must be enclosed in brackets - (even if there are no arguments, the brackets must still be present).
IN	number	The result of inputting at processor level from port x (0 <= x <= FFFFh). Loads the bc register pair with x and does the assembly language instruction <i>in a,(c)</i> .
INKEY\$	none	Reads the keyboard. The result is the character representing the key pressed if there is exactly one, else the empty string.
INT	number	Integer part (always rounds down).
LEN	string	Length.
LN	number	Natural logarithm (to base e). Error A if x <= 0.
NOT	number	0 if x > 0, 1 if x = 0. NOT has priority 4.
OR	binary operation, both operands numbers	$a \text{ OR } b \begin{cases} 1 & \text{if } b > 0 \\ a & \text{if } b = 0 \end{cases}$ <p>OR has priority 2.</p>
PEEK	number	The value of the byte in memory whose address is x (rounded to the nearest integer). Error B if x is not in the range 0 to 65535.
PI	none	π (3.1415927...).
POINT	Two arguments, x and y, both numbers; enclosed in brackets	1 if the pixel at (x,y) is ink colour. 0 if it is paper colour. Error B unless 0 <= x <= 255 and 0 <= y <= 175.

RND	none	The next pseudo-random number in a sequence generated by taking the powers of 75 modulo 65537, subtracting 1 and dividing by 65536. $0 \leq y < 1$.
SCREEN\$	Two arguments, x and y both numbers; enclosed in brackets	The character that appears, either normally or inverted, on the screen at line x, column y. Returns the empty string if the character is not recognised. Error B unless $0 < x \leq 23$ and $0 < y \leq 31$.
SGN	number	Sign of number. Returns -1 for negative, 0 for zero or +1 for positive.
SIN	number (in radians)	Sine x.
SQR	number	Square root. Error A if $x < 0$.
STR\$	number	The string of characters that would be displayed if x were printed.
TAN	number (in radians)	Tangent.
USR	number	Calls the machine code subroutine whose starting address is x. On return, the result is the contents of the bc register pair.
USR	string	The address of the bit pattern for the user-defined graphic corresponding to x. Error A if x is not a single letter between a and u, or a user-defined graphic.
VAL	string	Evaluates x (without its bounding quotes) as a numerical expression. Error C if x contains a syntax error, or gives a string value. Other errors possible, depending on the expression.
VAL\$	string	Evaluates x (without its bounding quotes) as a string expression. Error C if x contains a syntax error or gives a numeric value. Other errors possible (as for VAL).
-	number	Negation.

The following are binary operations:

+	Addition (on numbers), or concatenation (on strings)	
-	Subtraction	
*	Multiplication	
/	Division	
↑	Raising to a power. Error B if the left operand is negative	
=	Equals	} Both operands must be of the same type. The result is a number 1, if the comparison holds and 0 if it does not
>	Greater than	
<	Less than	
<=	Less than or equal to	
>=	Greater than or equal to	
<>	Not equal to	

Functions and operations have the following priorities:

OPERATION	PRIORITY
Subscripting and slicing	12
All functions except NOT and unary minus	11
↑	10
Unary minus (minus used to negate)	9
*, /	8
=, - (minus used to subtract)	6
=, >, <, <=, >=, <>	5
NOT	4
AND	3
OR	2

Statements

In this list:

- l represents a single letter.
- v represents a variable.
- x,y,z represent numerical expressions.
- m,n represent numerical expressions that are rounded to the nearest integer.
- e represents an expression.
- f represents a string valued expression.
- s represents a sequence of statements separated by colons.
- c represents a sequence of colour items, each terminated by commas or semicolons. A colour item has the form of a PAPER, INK, FLASH, BRIGHT, INVERSE, or OVER statement.

Note that arbitrary expressions are allowed everywhere (except for the line number at the beginning of a statement).

All statements except INPUT, DEF FN and DATA can be used either as commands or in programs (although they be more sensible in one than the other). A command or program line can have several statements, separated by colons. There is no restriction on whereabouts in a line any particular statement can occur; however, see IF and REM.

BEEP <i>x</i> , <i>y</i>	Sounds a note through the TV's speaker for <i>x</i> seconds at a pitch <i>y</i> semitones above middle C (or below middle C if <i>y</i> is negative).
BORDER <i>m</i>	Sets the border colour around the screen and also the paper colour for the lower part of the screen. Error K unless $0 < m <= 7$ (ie. unless <i>m</i> is not in the range 0 to 7).
BRIGHT	Sets brightness of characters subsequently printed; <i>n</i> =0 for normal, 1 for bright, 8 for transparent. Error K unless <i>n</i> is 0, 1 or 8.
CAT	Does not work without microdrive, etc.
CAT !	Gives a list of files currently resident on the silicon disc.
CIRCLE <i>x</i> , <i>y</i> , <i>z</i>	Draws an arc of a circle, centre (<i>x</i> , <i>y</i>) radius <i>z</i> .
CLEAR	Deletes all variables, freeing the space they previously occupied. Executes a RESTORE and CLS, resets the PLOT position to the bottom left-hand corner and clears the GO SUB stack.
CLEAR <i>n</i>	Like CLEAR, but if possible, changes the system variable RAMTOP to <i>n</i> and puts the new GO SUB stack there.
CLOSE #	Does not work without microdrive, etc.
CLS	(Clear screen). Clears the display file.
CONTINUE	Continues the program, starting where it left off last time it stopped with a report other than 0. If the report was 9 or L, then continues with the following statement (taking jumps into account), otherwise repeats the one where the error occurred. If the last report was in a command line then CONTINUE will attempt to continue the command line and will either go into a loop if the error was in 0 : 1, generate report 0 if it was in 0 : 2, or report N if it was in 0 : 3 or greater.
COPY	Sends a copy of the top 22 lines of display to the printer (if attached) in quad density Epson bit map format; otherwise does nothing. Report D if [BREAK] pressed.
DATA <i>e</i> ₁ , <i>e</i> ₂ , <i>e</i> ₃ ...	Part of the DATA list. Must be in a program; otherwise has no effect.
DEF FN <i>l</i> (<i>l</i> ₁ ,... <i>l</i> _{<i>k</i>}) = <i>e</i>	User-defined function definition. Must be in a program; otherwise has no effect. Each of <i>l</i> and <i>l</i> ₁ to <i>l</i> _{<i>k</i>} is either a single letter or a single letter followed by \$ for string argument or result. Takes the form DEF FN <i>l</i> () = <i>e</i> if no arguments.

DIM l (n ₁ ,...n _k)	Deletes any array with the name l, and sets up an array l of numbers with k dimensions n ₁ ,... n _k . Initialises all the values to 0.
DIM l\$ (n ₁ ,...n _k)	Deletes any array or string with the name l\$, and sets up an array l\$ of characters with k dimensions n ₁ ,... n _k . Initialises all the values to "". This can be considered as an array of strings of fixed length n _k , with k-1 dimensions (n ₁ ,... n _{k-1}). An array is undefined until it is dimensioned by DIM. Error 4 if there is no room to fit the array in.
DRAW x,y	DRAW x,y,0
DRAW x,y,z	Draws a line from the current plot position moving x horizontally and y vertically relative to it, while turning through angle z. Error B if line runs off the screen.
ERASE	Does not work without microdrive, etc.
ERASE ! f	Erase a file from the silicon disc.
FLASH	Defines whether characters will be flashing or steady; n=0 for steady, n=1 for flash, n=8 for no change.
FOR l=x TO y	FOR l=x TO y STEP 1
FOR l=x TO y STEP z	Deletes any simple variable l and sets up a control variable l with value x, limit y, step z, and looping address referring to the statement after the FOR statement. Checks if the initial value is greater (if step >= 0) or less (if step < 0) than the limit, and if so then skips to statement NEXT l, giving error 1 if there is none. See NEXT. Error 4 if there is no room for the control variable.
FORMAT f;n	Sets the baud rate of device f to baud rate n. Valid device "p" or "P" (the RS232), valid baud rates 75 to 19200.
GO SUB n	Pushes the line number of the GO SUB statement onto a stack; then as GO TO n. Error 4 can occur if there are not enough RETURNS.
GO TO n	Jumps to line n (or, if there is none, the first line after that).
IF x THEN s	If x is true (non-zero), then s is executed. Note that s comprises all the statements until the end of the line. The form 'IF x THEN line number' is not allowed.
INK n	Sets the ink (foreground) colour of characters subsequently printed; n is in the range 0 to 7 for a colour, n=8 for transparent or 9 for contrast. Error K unless 0 <= n <= 9.

INPUT...	<p>The '...' is a sequence of INPUT items, separated as in a PRINT statement by commas, semicolons or apostrophes. An INPUT item can be any of the following:</p> <ul style="list-style-type: none"> (i) Any PRINT item not beginning with a letter. (ii) A variable name. (iii) LINE, then a string type variable name. The PRINT items and separators in (i) are treated exactly as in PRINT, except that everything is printed in the lower part of the screen. For (ii) the computer stops and waits for input of an expression from the keyboard - the value of this is assigned to the variable. The input is echoed in the usual way and syntax errors give the flashing ?. For string type expressions, the input buffer is initialised to contain two string quotes (which can be erased if necessary). If the first character in the input is STOP ([SYMB SHIFT] A), then the program stops with error H. (iii) is like (ii) except that the input is treated as a string literal without quotes, and the STOP mechanism won't work - to stop it you must press cursor down \leftarrow instead.
INVERSE n	<p>Controls inversion of characters subsequently printed. If n=0, then characters are printed in normal video, as ink colour on paper colour. If n=1, characters are printed in inverse video, ie. paper colour on ink colour. Error K occurs (see part 28 of this chapter) if n is neither 0 nor 1.</p> <p>In 48 BASIC, pressing the [INV VIDEO] key is equivalent to INVERSE 1; pressing the [TRUE VIDEO] key is equivalent to INVERSE 0.</p>
LET v=e	<p>Assigns the value of e to the variable v. LET cannot be omitted. A simple variable is undefined until it is assigned to in either a LET, READ or INPUT statement. If v is a subscripted string variable, or a sliced string variable (substring), then the assignment is Procrustean (fixed length), ie. the string value of e is either truncated or filled out with spaces on the right, to make it the same length as specified in v.</p>
LIST	LIST 0.
LIST n	Lists the program to the upper part of the screen, starting at the first line whose number is at least n, and makes n the current line.
LLIST	LLIST 0.
LLIST n	Like LIST, but using the printer.
LOAD f	Loads the program and variables.
LOAD f DATA ()	Loads a numeric array.
LOAD f DATA \$()	Loads character array.
LOAD f CODE m,n	Loads (at most) n bytes, starting at address m.

LOAD f CODE m	Loads bytes starting at address m.
LOAD f CODE	Loads bytes back to the address from where they were saved.
LOAD f SCREEN\$	LOAD f CODE 16384,6912.
LOAD !	Like LOAD (for options, see above), but uses the silicon disc.
LPRINT...	Like PRINT, but using the printer.
MERGE f	Like LOAD f, but does not delete old program lines or variables, except to make way for new ones with the same line number or name.
MERGE ! f	Like MERGE f, but uses the silicon disc.
MOVE f ₁ ,f ₂	Does not work without the microdrive, etc.
NEW	Starts the BASIC system afresh, deleting any program and variables, and using the memory up to and including the byte whose address is in the system variable RAMTOP. The system variables UDG, P-RAMT, RASP and PIP are preserved. Returns control to the opening menu, but does not affect the silicon disc.
NEXT l	(i) Finds the control variable l. (ii) Adds its step to its value. (iii) If the step >=0 and the value > the limit; or if the step <0 and the value <the limit, then jumps to the looping statement. Error 2 if there is no variable l. Error 1 if variable l does not match control variable in FOR statement.
OPEN #	Does not work without the microdrive, etc.
OUT m,n	Outputs byte n at port m at processor level. (Loads the bc register pair with m, the a register with n, and does the assembly language instruction: out (c),a.) Error B unless 0 <= m <= 65535 and -255 <= n <= 255.
OVER n	Controls overprinting for characters subsequently printed. If n=0, characters obliterate previous characters at that position. If n=1, then new characters are mixed in with old characters to give ink colour wherever either (but not both) had ink colour, and paper colour where they were both paper or both ink. Error K unless n is 0 or 1.
PAPER n	Like INK, but controlling the paper (background) colour.
PAUSE n	Stops computing and displays the display file for n frames (at 50 frames per second - 60 frames per second in USA), or until a key is pressed. If n=0 then the pause is not timed, but lasts until a key is pressed. Error B unless 0 <= n <= 65535.

PLAY f (,f,f,f,...)

Interpret up to eight strings (see part 19 of this chapter) and play them simultaneously. The first three strings play via the TV speaker and (optionally) via the MIDI port; any subsequent strings can only be output via MIDI.

PLOT c; m, n

Prints an ink dot (subject to OVER and INVERSE) at the pixel (m,n), moving the PLOT position thereto.

Unless the colour items c specify otherwise, the ink colour at the character position containing the pixel is changed to the current permanent ink colour, and the others (paper colour, flashing and brightness) are left unchanged.

Error B unless $0 \leq m \leq 255$ and $0 \leq n \leq 175$.

POKE m, n

Writes the value n to the byte in store with address m.

Error B unless $0 \leq m \leq 65535$ and $-255 \leq n \leq 255$.

PRINT...

The '...' is a sequence of PRINT items, separated by commas, semicolons or apostrophes, and they are written to the display file for output to the screen.

A semicolon between two items has no effect - it is used purely to separate the items, a comma outputs the comma control character, and an apostrophe outputs the [ENTER] character (which is output by default if a PRINT statement does not end in a semicolon, comma or apostrophe).

A PRINT item can be:

(i) Empty, ie. nothing.

(ii) A numerical expression.

First a minus sign is printed if the value is negative. Now let x be the modulus of value. If $x \leq 10^{-5}$ or $x \geq 10^{13}$, then it is printed using scientific notation.

The mantissa part has up to eight digits (with no trailing zeros), and the decimal point (absent if only one digit) is after the first. The exponent part is E, followed by + or -, followed by one or two digits. Otherwise x is printed in ordinary decimal notation with up to eight significant digits, and no trailing zeros after the decimal point. A decimal point right at the beginning is always followed by a zero, so for instance .03 and 0.3 are printed as such. 0 is printed as a single digit 0.

(iii) A string expression.

The tokens in the string are expanded, possibly with a space before or after. Control characters have their control effect. Unrecognised characters print as ?.

(iv) AT m, n.

Outputs an AT control character followed by a byte for m (the line number) and a byte for n (the column number).

(v) TAB n.

Outputs a TAB control character followed by two bytes for n (least significant byte first) - the tab stop.

(vi) A colour item, which takes the form of a PAPER, INK, FLASH, BRIGHT, INVERSE or OVER statement.

RANDOMIZE	RANDOMIZE Ø.
RANDOMIZE n	Sets the system variable (called SEED) used to generate the next value of RND. If n < > 0, then SEED is given the value n. If n = 0 then SEED is given the value of another system variable (called FRAMES) that counts the frames so far displayed on the screen, and so should be fairly random. Error B unless 0 < = n < = 65535.
READ v ₁ , v ₂ , ... v _k	Assigns to the variable using successive expressions in the DATA list. Error C if an expression is the wrong type. Error E if there are variables left to be read when the DATA list is exhausted.
REM...	No effect. '...' can be any sequence of characters terminated by [ENTER]. No statements in the line will be acted upon after the REM, and colons will <i>not</i> be treated as separators.
RESTORE	RESTORE Ø.
RESTORE n	Restores the DATA pointer to the first DATA statement in line n. If line n doesn't exist (or is not a DATA statement), then the first DATA statement after line n is restored, and the next READ statement will start reading from there.
RETURN	Takes a reference to a statement off the GO SUB stack, and jumps to the line after it. Error 7 when there is no statement reference on the stack - (this probably means that there is some mistake in your program - ensure that all GO SUBs are balanced by RETURNS).
RUN	RUN Ø.
RUN n	CLEAR, and then GO TO n.
SAVE f	Saves the program and variables, giving it the name f. Error F if f is empty, or is greater than ten characters in length. See part 20 of this chapter.
SAVE f LINE m	Saves the program and variables so that if they are loaded, there is an automatic jump to line m.
SAVE f DATA ()	Saves the numeric array.
SAVE f DATA \$ ()	Saves the character array.
SAVE f CODE m, n	Saves n bytes starting at address m.
SAVE f SCREEN\$	SAVE f CODE 16384, 6912. Saves the current screen display.
SAVE ! f	Like SAVE, but operates with the silicon disc. See part 20 of this chapter.
SPECTRUM	Switches from 128 BASIC into 48 BASIC, maintaining any program in RAM. There is no switch back to 128 BASIC.
STOP	Stops the program with report 9. CONTINUE will resume the program at the following statement.
VERIFY	Like LOAD, but the information on cassette is not loaded into RAM - instead, it is just compared against what is already in RAM. Error R if the comparison shows different bytes.

Part 31

Example Programs

Programs...

Days
I Ching
Pangolins
Flag
Hangman

This section contains some example programs for your interest. If you wish to use the programs more than once, don't forget to SAVE them permanently onto cassette, or temporarily into the silicon disc.

Days

The first of these programs requires a date (in this century) to be input, and then gives the day of the week which corresponds to this date...

```
10 REM convert date to day
20 DIM d$(7,6): REM days of week
30 FOR n=1 TO 7: READ d$(n): NEXT n
40 DIM m(12): REM lengths of months
50 FOR n=1 TO 12: READ m(n): NEXT n
100 REM input date
110 INPUT "day?";day
120 INPUT "month?";month
130 INPUT "year (20th century only)?";year
140 IF year<1901 THEN PRINT "20th century starts at 1901":
    GO TO 100
150 IF year>2000 THEN PRINT "20th century ends at 2000":
    GO TO 100
160 IF month<1 THEN GO TO 210
170 IF month>12 THEN GO TO 210
180 IF year/4-INT(year/4)=0 THEN LET m(2)=29: REM leap year
190 IF day>m(month) THEN PRINT "This month has only ";
    m(month);" days.": GO TO 500
200 IF day>0 THEN GO TO 300
210 PRINT "Stuff and nonsense. Give me a real date."
220 GO TO 500
```

...continued on next page

```
300 REM convert date to number of days since start of century
310 LET y=year-1901
320 LET b=365*y+INT (y/4): REM number of days to start of year
330 FOR n=1 TO month-1: REM add on previous months
340 LET b=b+m(n): NEXT n
350 LET b=b+day
400 REM convert to day of week
410 LET b=b-7* INT (b/7)+1
420 PRINT day;" ";month;" ";year
430 FOR n=6 TO 3 STEP -1: REM remove trailing spaces
440 IF d$(b,n) <> " " THEN GO TO 460
450 NEXT n
460 LET e$=d$(b, TO n)
470 PRINT" is a "; e$; "day"
500 LET m(2)=28: REM restore February
510 INPUT "again?", a$
520 IF a$="n" THEN GO TO 540
530 IF a$ <> "N" THEN GO TO 100
540 STOP
1000 REM days of week
1010 DATA "Mon", "Tues", "Wednes"
1020 DATA "Thurs", "Fri", "Satur", "Sun"
1100 REM lengths of months
1110 DATA 31, 28, 31, 30, 31, 30
1120 DATA 31, 31, 30, 31, 30, 31
```

I Ching

Here is a program to throw coins for the 'I Ching'. The patterns it produces are 'upside down' -however, the results should still prove acceptable...

```
5 RANDOMIZE
10 FOR m=1 TO 6: REM for 6 throws
20 LET c=0: REM initialize coin total to 0
30 FOR n=1 TO 3: REM for 3 coins
40 LET c=c+2+INT (2*RND)
50 NEXT n
60 PRINT "  ";
70 FOR n=1 TO 2: REM 1st for the thrown hexagram, 2nd for
   the changes
80 PRINT "----";
90 IF c=7 THEN PRINT "--";
100 IF c=8 THEN PRINT " ";
110 IF c=6 THEN PRINT "X";: LET c=7
120 IF c=9 THEN PRINT "0";: LET c=8
130 PRINT "---- ";
140 NEXT n
150 PRINT
160 INPUT a$
170 NEXT m: NEW
```

After you have typed in this program, RUN it, then press **[ENTER]** five times to get the two hexagrams. Look these up in a copy of the Chinese Book of Changes. The text will describe a situation and the courses of action appropriate to it, and you must ponder deeply to discover the parallels between that and your own life. Press **[ENTER]** a sixth time, and the program will erase itself - this is to discourage you from using it frivolously!

Many people find the texts are always more apt than they would expect on grounds of chance; this may or may not be the case with your **+2**. *In general, computers are pretty godless creatures.*

Pangolins

Here is a program to play 'pangolins'. You think of an animal, and the computer tries to guess what it is by asking you questions that can be answered either 'yes' or 'no'. If it has never heard of your animal before, then it asks you to type in a question that it can use next time to find out whether someone's given it your new animal...

```
5 REM pangolins
10 LET nq=100: REM number of questions and animals
15 DIM q$(nq,50): DIM a(nq,2): DIM r$(1)
20 LET qf=8
30 FOR n=1 TO qf/2-1
40 READ q$(n): READ a(n,1): READ a(n,2)
50 NEXT n
60 FOR n=n TO qf-1
70 READ q$(n): NEXT n

100 REM start playing
110 PRINT "Think of an animal.,""Press any key to continue."
120 PAUSE 0
130 LET c=1: REM start with 1st question
140 IF a(c,1)=0 THEN GO TO 300
150 LET p$=q$(c): GO SUB 910
160 PRINT "?": GO SUB 1000
170 LET i =1: IF r$="y" THEN GO TO 210
180 IF r$="Y" THEN GO TO 210
190 LET i =2: IF r$="n" THEN GO TO 210
200 IF r$<>"N" THEN GO TO 150
210 LET c=a(c,i ): GO TO 140

300 REM animal
310 PRINT "Are you thinking of"
320 LET p$=q$(c): GO SUB 900: PRINT "??"
330 GO SUB 1000
340 IF r$="y" THEN GO TO 400
350 IF r$="Y" THEN GO TO 400
360 IF r$="n" THEN GO TO 500
370 IF r$="N" THEN GO TO 500
380 PRINT "Answer me properly when I'm","talking to you.": GO
    TO 300

400 REM guessed it
410 PRINT "I thought as much.": GO TO 800
```

...continued on next page

```

500 REM new animal
510 IF qf>nq-1 THEN PRINT "I'm sure your animal is very",
    "interesting, but I don't have","room for it just now.": GO TO 800
520 LET q$(qf)=q$(c): REM move old animal
530 PRINT "What is it, then?": INPUT q$(qf+1)
540 PRINT "Tell me a question which dist-","inguishes
    between "
550 LET p$=q$(qf): GO SUB 900: PRINT " and"
560 LET p$=q$(qf+1): GO SUB 900: PRINT " "
570 INPUT s$: LET b=LEN s$
580 IF s$(b)="?" THEN LET b=b-1
590 LET q$(c)=s$(TO b): REM insert question
600 PRINT "What is the answer for"
610 LET p$=q$(qf+1): GO SUB 900: PRINT "?"
620 GO SUB 1000
630 LET i =1: LET io=2: REM answers for new and old animals
640 IF r$="y" THEN GO TO 700
650 IF r$="Y" THEN GO TO 700
660 LET i =2: LET io=1
670 IF r$="n" THEN GO TO 700
680 IF r$="N" THEN GO TO 700
690 PRINT "That's no good. ": GO TO 600

700 REM update answers
710 LET a(c,i )=qf+1: LET a(c,io)=qf
720 LET qf=qf+2: REM next free animal space
730 PRINT "That fooled me."

800 REM again?
810 PRINT "Do you want another go?": GO SUB 1000
820 IF r$="y" THEN GO TO 100
830 IF r$="Y" THEN GO TO 100
840 STOP

900 REM print without trailing spaces
905 PRINT " ";
910 FOR n=50 TO 1 STEP -1
920 IF p$(n)<>" " THEN GO TO 940
930 NEXT n
940 PRINT p$(TO n): RETURN

```

...continued on next page

```
1000 REM get reply
1010 INPUT r$: IF r$="" THEN RETURN
1020 LET r$=r$(1): RETURN

2000 REM initial animals
2010 DATA "Does it live in the sea",4,2
2020 DATA "Is it scaly",3,5
2030 DATA "Does it eat ants",6,7
2040 DATA "a whale", "a blanchmange", "a pangolin", "an ant"
```

Flag

Here is a program to draw a Union Jack...

```
5 REM union flag
10 LET r=2: LET w=7: LET b=1
20 BORDER 0: PAPER b: INK w: CLS
30 REM black in bottom of screen
40 INVERSE 1
50 FOR n=40 TO 0 STEP -8
60 PLOT PAPER 0;7,n: DRAW PAPER 0;241,0
70 NEXT n: INVERSE 0
100 REM draw in white parts
105 REM St. George
110 FOR n=0 TO 7
120 PLOT 104+n,175: DRAW 0,-35
130 PLOT 151-n,175: DRAW 0,-35
140 PLOT 151-n,48: DRAW 0,35
150 PLOT 104+n,48: DRAW 0,35
160 NEXT n
200 FOR n=0 TO 11
210 PLOT 0,139-n: DRAW 111,0
220 PLOT 255,139-n: DRAW -111,0
230 PLOT 255,84+n: DRAW -111,0
240 PLOT 0,84+n: DRAW 111,0
250 NEXT n
```

...continued on the next page

```
300 REM St. Andrew
310 FOR n=0 TO 35
320 PLOT 1+2*n,175-n: DRAW 32,0
330 PLOT 224-2*n,175-n: DRAW 16,0
340 PLOT 254-2*n,48+n: DRAW -32,0
350 PLOT 17+2*n,48+n: DRAW 16,0
360 NEXT n
370 FOR n=0 TO 19
380 PLOT 185+2*n,140+n: DRAW 32,0
390 PLOT 200+2*n,83-n: DRAW 16,0
400 PLOT 39-2*n,83-n: DRAW 32,0
410 PLOT 54-2*n,140+n: DRAW -16,0
420 NEXT n
425 REM fill in extra bits
430 FOR n=0 TO 15
440 PLOT 255,160+n: DRAW 2*n-30,0
450 PLOT 0,63-n: DRAW 31-2*n,0
460 NEXT n
470 FOR n=0 TO 7
480 PLOT 0,160+n: DRAW 14-2*n,0
485 PLOT 255,63-n: DRAW 2*n-15,0
490 NEXT n
500 REM red stripes
510 INVERSE 1
520 REM St. George
530 FOR n=96 TO 120 STEP 8
540 PLOT PAPER r;7,n: DRAW PAPER r;241,0
550 NEXT n
560 FOR n=112 TO 136 STEP 8
570 PLOT PAPER r;n,168: DRAW PAPER r;0,-113
580 NEXT n
600 REM St. Patrick
610 PLOT PAPER r;170,140: DRAW PAPER r;70,35
620 PLOT PAPER r;179,140: DRAW PAPER r;70,35
630 PLOT PAPER r;199,83: DRAW PAPER r;56,-28
640 PLOT PAPER r;184,83: DRAW PAPER r;70,-35
650 PLOT PAPER r;86,83: DRAW PAPER r;-70,-35
660 PLOT PAPER r;72,83: DRAW PAPER r;-70,-35
670 PLOT PAPER r;56,140: DRAW PAPER r;-56,28
680 PLOT PAPER r;71,140: DRAW PAPER r;-70,35
690 INVERSE 0: PAPER 0: INK 7
```

If you aren't British, have a go at drawing your own flag. Tricolours are fairly easy, although some of the colours - for instance the orange in the Irish flag - might present difficulties. If you're trying to create the stars and stripes in the flag of the USA, you might be able to fit the * characters in. Once you've drawn a flag, you could store it away in the silicon disc using SAVE ! "f l a g" SCREEN\$, and then draw a different flag and save it under a different name. There's room for about 10 different screens in the silicon disc, so you could put on quite a varied display.

Hangman

Here is a program to play hangman. In case you're not familiar with the game - one player enters a word, and the other player tries to guess it...

```
5 REM Hangman
10 REM set up screen
20 INK 0: PAPER 7: CLS
30 LET x=240: GO SUB 1000: REM draw man
40 PLOT 238,128: DRAW 4,0: REM mouth
100 REM set up word
110 INPUT w$: REM word to guess
120 LET b=LEN w$: LET v$=""
130 FOR n=2 TO b: LET v$=v$+" "
140 NEXT n: REM v$=word guessed so far
150 LET c=0: LET d=0: REM guess & mistake counts
160 FOR n=0 TO b-1
170 PRINT AT 20,n;"-";
180 NEXT n: REM write -'s instead of letters
200 INPUT "Guess a letter: ";g$
210 IF g$="" THEN GO TO 200
220 LET g$=g$(1): REM 1st letter only
230 PRINT AT 0,c:g$
240 LET c=c+1: LET u$=v$
250 FOR n=1 TO b: REM update guessed word
260 IF w$(n)=g$ THEN LET v$(n)=g$
270 NEXT n
280 PRINT AT 19,0:v$
290 IF v$=w$ THEN GO TO 500: REM word guessed
300 IF v$<>u$ THEN GO TO 200: REM guess was right
400 REM draw next part of gallows
410 IF d=8 THEN GO TO 600: REM hanged
420 LET d=d+1
430 READ x0,y0,x,y
440 PLOT x0,y0: DRAW x,y
450 GO TO 200
```

...continued on next page

```
500 REM free man
510 OVER 1: REM rub out man
520 LET x=240: GO SUB 1000
530 PLOT 238,128: DRAW 4,0: REM mouth
540 OVER 0: REM redraw man
550 LET x=146: GO SUB 1000
560 PLOT 143,129: DRAW 6,0, PI/2: REM smile
570 GO TO 800
600 REM hang man
610 OVER 1: REM rub out floor
620 PLOT 255,65: DRAW -48,0
630 DRAW 0,-48: REM open trapdoor
640 PLOT 238,128: DRAW 4,0: REM rub out mouth
650 REM move limbs
655 REM arms
660 PLOT 255,117: DRAW -15,-15: DRAW -15,15
670 OVER 0
680 PLOT 236,81: DRAW 4,21: DRAW 4,-21
690 OVER 1: REM legs
700 PLOT 255,66: DRAW -15,15: DRAW -15,-15
710 OVER 0
720 PLOT 236,60: DRAW 4,21: DRAW 4,-21
730 PLOT 237,127: DRAW 6,0, -PI/2: REM frown
740 PRINT AT 19,0:w$
800 INPUT "again? ";a$
810 IF a$="" THEN GO TO 850
820 LET a$a$(1)
830 IF a$="n" THEN STOP
840 IF a$(1)="N" THEN STOP
850 RESTORE : GO TO 5
1000 REM draw man at column x
1010 REM head
1020 CIRCLE x,132,8
1030 PLOT x+4,134: PLOT x-4,134: PLOT x,131
1040 REM body
1050 PLOT x,123: DRAW 0,-20
1055 PLOT x,101: DRAW 0,-19
1060 REM legs
1070 PLOT x-15,66: DRAW 15,15: DRAW 15,-15
```

...continued on next page

1080 REM arms

1090 PLOT x-15,117: DRAW 15,-15: DRAW 15,15

1100 RETURN

2000 DATA 120,65,135,0,184,65,0,91

2010 DATA 168,65,16,16,184,81,16,-16

2020 DATA 184,156,68,0,184,140,16,16

2030 DATA 204,156,-20,-20,240,156,0,-16

Part 32

Binary and hexadecimal

Subjects covered...

Number systems
Bits and bytes

This section describes how computers count, using the binary system.

Most European languages count using a more or less regular pattern of tens - in English, for example, although it starts off a bit erratically, it soon settles down into regular groups...

twenty, twenty one, twenty two,... twenty nine
thirty, thirty one, thirty two,... thirty nine
forty, forty one, forty two,... forty nine

...and so on, and this is made even more systematic with the numerals that we use. However, the only reason for using ten (the *decimal* system) is that we happen to have ten fingers and thumbs.

Instead of using the decimal system - based on ten, computers use a form of binary called *hexadecimal* (or 'hex' for short) which is based on sixteen. As there are only ten digits available in our number system we need six extra digits to do the counting. So we use A, B, C, D, E and F. And what comes after F? Well, just as we, with ten fingers, write 10 for ten (a hand full), so computers use 10 for sixteen. Comparing counting in decimal to hex...

DECIMAL	HEX
----------------	------------

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

...continued...

25	19
26	1A
27	1B
...etc...	
31	1F
32	20
33	21
...etc...	
158	9E
159	9F
160	A0
161	A1
...etc...	
255	FF
256	100

...and so on.

If you are using hex notation and you want to make the fact quite plain, then write 'h' at the end of the number, and say 'hex'. For instance, for one hundred and fifty eight (decimal), write '9Eh' and say 'nine E hex'.

You may be wondering what all this has to do with computers. In fact, computers behave as though they had only two digits, represented by a low voltage (or off) known as 0, and a high voltage (or on) known as 1. This is called the *binary* system, and the two binary digits are called *bits* - so a bit is either 0 or 1.

So to expand the previous table of counting to include binary...

DECIMAL	HEX	BINARY
0	0	0
1	1	1
2	2	10
3	3	11
4	4	100
5	5	101
6	6	110
7	7	111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111
16	10	10000
17	11	10001
...etc...		

It is customary to 'pad out' binary numbers with leading zeros so that they always contain at least four bits - for example, 0000, 0001, 0010, 0011 (representing 0 to 3 decimal).

Converting between binary and hex is very easy (use the previous table to help you):

To convert a binary number to hex, split the binary number into groups of four bits (starting at the *right* of the number) and convert each group into its corresponding hex digit. Finally, put the hex digits together to form the complete hex number. For example, to convert 10110100 binary into hex, convert the first (right hand) group of four bits (0100) to 4 hex, then convert the next group of four bits (1011) to B hex, put them together, and you have the complete hex number - B4h. If the binary number is longer than eight bits, you can continue converting each group of four bits into one hex digit. For example, 11101011110000 binary corresponds to 3AF0h.

To convert a hex number to binary, change each hex digit into four bits (again, starting at the right) then put the bits together to form the complete binary number. For example, to convert F3h to binary, first convert 3 which corresponds to 0011 binary (remember - you must use zeros to make the binary number four bits' long), then convert F which corresponds to 1111 binary, put them together, and you have the complete binary number - 11110011.

Although computers use a pure binary system, humans often write the numbers stored inside a computer using hex notation - after all, the number 3AF0h (for example) is far more likely to be easily and correctly read than 0011101011110000 in sixteen bit binary notation.

The bits inside the computer are mostly grouped into sets of eight, or *bytes*. A single byte can represent any number from 0 to 255 decimal (11111111 binary or FFh).

Two bytes can be grouped together to make what is technically called a *word*. A word can be expressed using sixteen bits or four hex digits, and represents a number from 0 to 65535 decimal (1111111111111111 binary or FFFFh).

A byte is *always* eight bits, but words vary in length from computer to computer.

The BIN notation used in part 14 of this chapter provides a means of entering numbers in binary on the +2, ie. BIN 10 represents 4 decimal, BIN 111 represents 7 decimal, BIN 11111111 represents 255 decimal, and so on.

You can only use 0s and 1s for this, so the number must be a non-negative whole number - for instance, you can *not* use BIN -11 to represent -3 decimal, but you can use -BIN 11 instead. The number must also be no greater than decimal 65535 - ie. it can't have more than sixteen bits. If you pad out a binary number with leading zeros, for example, BIN 000000001, BIN will rightly ignore them and treat the number as if it were BIN 1.

Chapter 9

Using the calculator

Subjects covered...

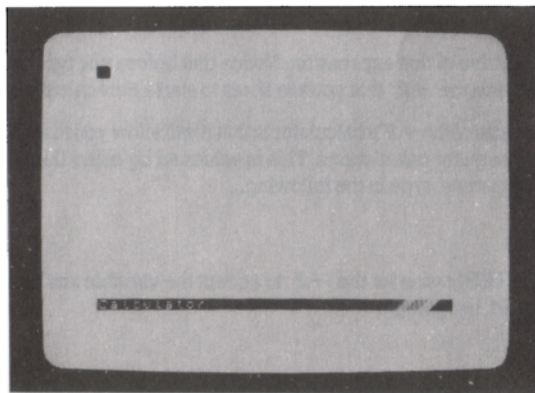
- Selecting the calculator
- Entering numbers
- Running total
- Using built-in mathematical functions
- Editing the screen
- Assigning variables
- Exit-ing from the calculator

The **+2** can be used as a full function calculator.

To use the calculator, call up the opening menu and select the option 'C a l c u l a t o r'. (If you don't know how to select a menu option, refer back to chapter 2.)

The calculator may be selected as soon as the **+2** is switched on. Alternatively, if you are working on a 128 BASIC program, you may select the calculator by choosing the 'E x i t' option from the edit menu (which returns you to the opening menu), at which point you can select the 'C a l c u l a t o r' option. Note that any BASIC program which was being worked on when you selected the calculator, will be remembered and restored when you exit from the calculator.

When you have selected the 'C a l c u l a t o r' option, the screen will change to...



...and the **+2**'s calculator is ready to accept your first entry. Type in...

6+4

As soon as you press **[ENTER]**, the answer 10 will appear. (Note that you *don't* key in = as you would on a conventional calculator.)

You will see that the cursor is positioned to right of the answer, which is a *running total* (like on a conventional calculator). This means that you can simply type in the next operation to be carried out on the running total (without having to type in a whole new calculation). So, with the cursor still positioned to the right of the 10 on the screen, type in...

/5

...and back comes the answer 2. Now type in...

*PI

This produces the result 6.2831853 on the screen. The +2 has used its built-in π function - all you had to do was type in PI. This applies to *all* the +2's mathematical functions. To demonstrate, type in...

*ATN 60

...which gives the result 9.7648943. You may also 'edit' the contents of the screen. To demonstrate, move the cursor (using the cursor left key \leftarrow) to the beginning of the line and then type in 'INT' so that the line reads...

INT 9.7648943

...and as soon as **[ENTER]** is pressed, back comes the answer 9. This also demonstrates that the +2 doesn't *have to* perform a calculation in order to print the value of an expression. As another example, press **[ENTER]** then type...

1E6

...and back will come the value of that expression. Notice that before you typed in '1E6', you pressed **[ENTER]** on its own - this tells the +2 that you are about to start a new calculation.

One extremely useful feature of the +2's calculator is that it will allow you to assign values to variables and then use them in subsequent calculations. This is achieved by using the LET statement (as you would in BASIC). To demonstrate, type in the following...

LET x=10

(You must then press **[ENTER]** *twice* for the +2 to accept the variable assignment.) Now verify that the variable x is being used, by typing...

x+90

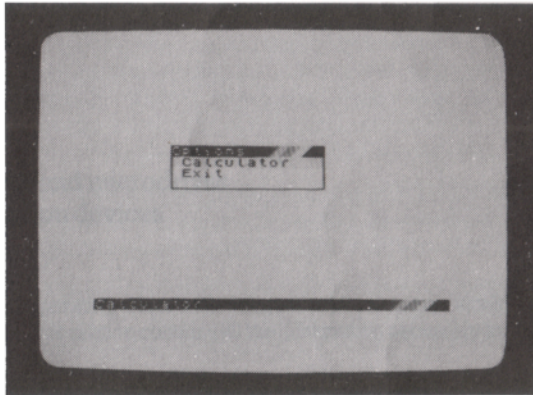
...then...

+x*x

If you are using the calculator whilst working on a BASIC program, then any variables used by the calculator should be chosen so that they do *not* conflict with those used by the program itself.

BASIC keywords are *not* allowed to be used as variable names.

When you have finished using the calculator, press the [EDIT] key. The screen will change to...



Select the 'E x i t' option to return to the opening menu. If you were working on a 128 BASIC program before you started using the calculator, then you may return to the program by selecting the option '128 BASIC'. (If you wish to continue using the calculator, then select the 'Calculator' option.)

Chapter 10

Connecting peripherals to your +2

Subjects covered...

Joystick(s)
VDU Monitor
Amplifier
Printer
Serial devices
MIDI device
Keypad
Interface One and microdrives
Other expansion devices

The +2 is capable of operating with a wide range of add-ons (*peripherals*) such as joystick(s), VDU monitor, amplifier, etc. This section contains all the information necessary to connect these.

Joystick(s)

We recommend that you use the Sinclair **SJS1** joystick(s) with the +2. Any other type of joystick (eg. Atari) will *not* operate directly, as its connecting plug is wired differently.

There are two joystick sockets at the left hand side of the +2. In general, games use the **JOYSTICK 1** socket.

If a program offers you a *choice* of joystick types, then choose the 'Interface Two' (or 'Sinclair') option (as the +2's joystick circuitry is designed to work exactly like the Interface Two).

It is safe to plug in (or unplug) a joystick while the +2 is switched on.

PIN	FUNCTION
1	not used
2	ground
3	not used
4	fire
5	up
6	right
7	left
8	ground
9	down

JOYSTICK 1 and JOYSTICK 2 sockets:



VDU Monitor

The +2 can use a monochrome or colour VDU monitor instead of (or in addition to) a TV. If the monitor that you wish to use isn't advertised as being Spectrum +2 (or Spectrum 128) compatible, then the chances are you'll have to buy a lead for it (contact your Sinclair dealer).

Note that unless your monitor accepts a *BRIGHT* signal it will only display 8 of the 16 available colours.

RGB socket:



PIN	SIGNAL	LEVEL
1	composite PAL	1.2V pk-pk/75 ohms
2	0 volts	-
3	bright	TTL
4	composite sync	TTL
5	vertical sync	TTL
6	green	TTL
7	red	TTL
8	blue	TTL

When using a monitor, some provision may have to be made for sound (if required). If the monitor has an audio input, then this should be connected to the **SOUND** socket at the back of the **+2**; if the monitor is *not* capable of producing sound, then an external amplifier will have to be used. See the next paragraph for further details.

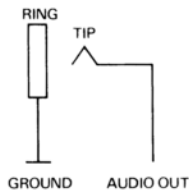
Amplifier

The **+2** normally reproduces sound through the TV set it is connected to. However, if a VDU monitor is being used, or if you would like to record or amplify the sound further, then a sound signal is available from the **SOUND** socket at the back of the **+2**. This is a 3.5mm mono jack socket producing 200mV pk-pk at approximately 5kohms impedance. When using an amplifier, it is worth remembering that the datacorder's 'load' and 'save' signals are also fed to the **SOUND** socket (and therefore the amplifier's volume control should be turned down when performing these operations).

Another point to note is that the level of sound produced by **BEEP** is set to be the same as that of *all three* channels of **PLAY** running at the same time. In practice, this means that **BEEP** will sound quite a lot louder than **PLAY** (which may cause problems if sound levels are critical).

It is safe to plug in (or unplug) an amplifier, tape recorder, etc. into the **SOUND** socket while the **+....2** is switched on.

SOUND socket:



Printer (and other serial devices)

The **+2** may be used with most serial printers conforming to the RS232 standard. It is recommended that inexperienced users should *not* attempt to experiment with interface connections. You should obtain a suitable computer-to-printer lead from your Sinclair dealer, and you should always follow the printer manufacturer's connection and operating instructions.

The printer should be connected to the **RS232/MIDI** socket at the rear of the **+2**.

To connect any serial device to the **+2**, you will require a Spectrum **+2** serial lead - available from your Sinclair dealer. If you wish to wire-up your own, then the connections are as follows (on the next page)...

PIN	FUNCTION
1	GND
2	TXD
3	RXD
4	DTR
5	CTS
6	+12V

RS232 socket:



MIDI device

Although the +2's **MIDI** (Musical Instrument Digital Interface) port shares the same socket as the **RS232**, you will need a different lead for it (available from your Sinclair dealer). The lead should be connected into the 'MIDI IN' socket on your synthesiser, drum machine, etc. There is *no* provision for the +2 to receive MIDI data - it can only act as a *source*. No setting up of the MIDI is necessary before use (except the commands within **P L A Y** to turn it on).

Using the MIDI interface will not disturb the RS232's baud rate setting.

MIDI socket:



PIN	FUNCTION
1	RETURN
2	not used
3	not used
4	not used
5	DATA OUT
6	not used

Keypad

The keypad (check availability with your Sinclair dealer) offers access to a wide range of editing facilities such as 'move by page', 'delete by word' and 'delete to end of line'. It may also be used as a calculator keyboard.

The keypad should be connected to the **KEYPAD** socket at the rear of the **+2**.

Interface One and microdrives

The **+2** will work with the 'Interface One' and with microdrives. Full instructions for use come with these, and they are available from your Sinclair dealer.

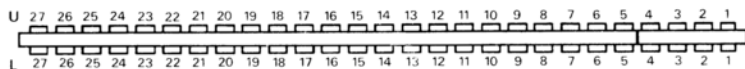
The 'Interface One' and microdrives are connected to the **EXPANSION I/O** socket at the rear of the **+2**.

Other expansion devices

The **+2** can connect to a very wide range of peripherals via the **EXPANSION I/O** socket at the rear of the machine. Although this socket is much the same as on the old-style Spectrum 48K, there is no guarantee that a device which ran correctly on a Spectrum 48K will run on a **+2**. You should, therefore, before you purchase any expansion device or add-on, verify that it will work with the **+2**, and not just with a 48K Spectrum.

<p>WARNING - It is very dangerous indeed to plug in (or unplug) any device from the EXPANSION I/O socket while the +2 is switched on - you will probably damage both the +2 and the expansion device if you do so.</p>
--

EXPANSION I/O socket:



PIN	UPPER ROW (U)	LOWER ROW (L)
1	A15	A14
2	A13	A12
3	D7	+5V
4	not used	+9V
5	D0	0V
6	D1	0V
7	D2	CK
8	D6	A0
9	D5	A1
10	D3	A2
11	D4	A3
12	$\overline{\text{INT}}$	$\overline{\text{IORQGE}}$
13	$\overline{\text{NMI}}$	0V
14	$\overline{\text{HALT}}$	not used
15	$\overline{\text{MREQ}}$	not used
16	$\overline{\text{IORQ}}$	not used
17	$\overline{\text{RD}}$	not used
18	$\overline{\text{WR}}$	$\overline{\text{BUSRQ}}$
19	-5V	$\overline{\text{RESET}}$
20	$\overline{\text{WAIT}}$	A7
21	+12V	A6
22	-12V	A5
23	$\overline{\text{M1}}$	A4
24	$\overline{\text{RFSH}}$	$\overline{\text{ROMCS}}$
25	A8	BUSACK
26	A10	A9
27	not used	A11

Index

A

ABS 66, 170
ACS 75, 170
Aerial lead 7, 8
Amplifier 116, 199
AND 82, 170
Animation 131
Apostrophe 44, 178
Arithmetical expressions 58, 67, 70, 173
Arrays 79, 129, 144, 146, 168
ASN 75, 170
Assembler 153, 155
AT 94, 96, 97, 136, 150, 178
ATN 75, 170
ATTR 103, 105, 170
Attributes 101, 103, 142, 152

B

BASIC 21, 23, 29, 37, 152, 195
Baud rate 135
BEEP 117, 174, 199
BIN 89, 170, 192
Binary 89, 190
Bit 191
BORDER 105, 174
Brackets 58, 63, 67, 68, 129, 170
[BREAK] key 40, 46, 136, 163
BRIGHT 102, 174
Brightness 101
Byte 89, 140, 192

C

Calculator 193
[CAPS LOCK] key 32, 38
[CAPS SHIFT] key 30, 32, 38, 114
Cassette operation 125, 132, 164
CAT 131, 174
Characters 85, 90, 155
CHRS 85, 99, 170
CIRCLE 109, 174
Circles 73, 108
CLEAR 146, 153, 174
CLOSE 174
CLS 48, 96, 174
C mode 32
CODE 85, 130, 154, 164, 171
Colon 45, 178

Colour 11, 100, 110, 150, 152, 178
Comma 44, 178
Commands 37
Connections 8, 197
Contents 4
CONTINUE 44, 46, 151, 161, 174
Contrast 102
Control codes/characters 90, 155
Control variable 51, 145
Coordinates 95, 107
COPY 136, 174
COS 74, 171
Cursor 23, 30, 39, 42, 60, 150

D

DATA 56, 129, 150, 163, 174
Datacoder 17, 20, 125, 132, 164
DC 9V socket 8
Decimal 190
DEF 67, 164, 174
Degrees 75
[DELETE] key 26
DIM 79, 175
Dimensions 79, 175
Dots 101
DRAW 108, 175

E

[EDIT] key 16, 24, 43, 195
Editing 23, 26, 28, 35, 39, 40
E mode 32
Empty string 60, 95, 114, 128, 169
[ENTER] key 44, 114
ERASE 131, 175
Error messages 161
EXP 71, 171
EXPANSION I/O socket 139, 201, 202
Exponents 59, 70
[EXTEND MODE] key 32, 38

F

FLASH 102, 175
Flashing 101
FN 67, 164, 171
FOR 51, 161, 175
FORMAT 135, 165, 175
Functions 64, 67, 170

G

Gmode 34
GO SUB 54, 162, 175
GO TO 28, 44, 46, 175
[GRAPH] key 34, 39, 86
Graphics 86, 107

H

Hardware 140, 166
Headphones 116
Hexadecimal 190

I

IF 48, 82, 175
IN 138, 171
INK 102, 175
INKEY\$ 114, 171
INPUT 43, 97, 176
Instructions 37
INT 66, 171
Interface one 201
Interface two 197
[INV VIDEO] key 176
INVERSE 103, 110, 176
I/O ports 138, 166, 197

J

Joysticks 139, 197
JOYSTICK sockets 197, 198

K

Keyboard 30, 37, 114, 139
Keypad 137, 139, 201
KEYPAD socket 166, 201
Keywords 41, 59, 195
K mode 30, 35

L

LEFT\$ 68
LEN 64, 171
LET 41, 58, 176, 194
LINE 98, 129, 154
Line numbers 24, 26, 41
LIST 40, 42, 176
Listing 25, 40, 42
LLIST 135
L mode 31
LN 72, 171

LOAD 127, 129, 133, 154
Loading a program 17, 19, 127, 128
Logarithmic function 72
Logical expressions 82
Loop 50
LPRINT 135, 151

M

Machine code 153
Mains plug 8
Mathematical expressions 58, 67, 70, 173
Memory 138, 140, 147, 166
Menus 14, 16, 17, 19, 24
MERGE 128, 134, 164
Messages 161
Microdrive 137, 143, 201
MID\$ 68
MIDI 123, 139, 177, 200
MIDI socket 166, 200
Monitor 116, 198
Motion 112
MOVE 177
Music 116, 124

N

Nesting 52
Network 137
NEW 43, 146, 177
NEXT 51, 161, 177
NOT 82, 171
Null string 60, 95, 114, 128, 169
Numerical expressions 59, 66, 143, 168

O

OPEN 177
OR 82, 171
OUT 138, 177
OVER 104, 110, 177
Overprinting 96, 104, 106, 110

P

PAPER 102, 177
PAUSE 112, 177
PEEK 90, 112, 141, 148, 171
Peripherals 137, 197
PI 72, 109, 171
Pixel 95, 101, 107, 109
PLAY 30, 116, 124, 165, 177, 199

PLOT	107, 178	SGN	66, 172
POINT	109, 171	Shift keys	30, 38
POKE	90, 99, 141, 148, 178	Sign	66
Ports	197	Silicon disc	30, 131
Power supply unit	7, 8	SIN	74, 107, 172
Precautions	3	Slicing	61, 81, 168
PRINT	41, 44, 60, 94, 151, 178	Software	1, 17, 19, 127
Printer	25, 135, 199	Sound	116, 166, 199
Processor	153, 166	SOUND socket	116, 199
Procrustean assignment	62, 80	Speakers	116
Pseudo-random	77	SPECTRUM	30, 148, 179
Q		SQR	67, 107, 172
Quotes	45, 60, 65	Square root	67
R		Stack	54, 146
Radians	75	STEP	51, 175
RAM	138, 140, 147, 152, 154, 166	STOP	49, 162, 179
RAMTOP	146, 152, 164, 174	Stopping a program	44, 46, 49
RANDOMIZE	78, 151, 179	STR\$	65, 172
Random numbers	77	String expressions. 45, 59, 60, 61, 64, 80, 91, 145	
READ	56, 163, 179	Subroutine	54
Relational operators	48, 91, 173	Subscript	79, 162
REM	43, 179	Substring	61, 169
Renumbering	24, 43, 149	Switching on/off	11, 16
Reports	161	[SYMB SHIFT] key	30, 38
RESET button	14, 18	Syntax error	35
Resetting the computer	18	System variables	143, 148
RESTORE	57, 179	T	
RETURN	54, 162, 179	TAB	96, 136, 150, 178
RGB socket	198	TAN	75, 172
RIGHT\$	68	Test signal	11
RND	77, 151, 172	THEN	48, 82, 175
ROM	138, 140, 147, 166	TL\$	68
Roots	70	TO	51, 61, 175
Rounding numbers	66, 68	Tokens	30, 85, 90, 158
RS232	135, 137, 139, 199	Trigonometrical functions	73
RS232 socket	166, 199	Troubleshooting	13
RUN	27, 44, 46, 127, 179	[TRUE VIDEO] key	176
S		Tuning in TV	11
SAVE	125, 129, 132, 154	TV	8, 11, 100, 103, 112
Saving a program	125, 129	TV socket	8
Screen display	12, 25, 27, 35, 40, 95, 130	U	
SCREEN\$	94, 130, 172	Unpacking	7
Scroll	40, 96, 99, 151	User defined graphics	34, 39, 88, 92, 111
Semicolon	44, 178	USR	89, 111, 148, 154, 172
Setting up	8, 9		

V

VAL 65, 172
VAL\$ 66, 172
Variables 59, 68, 79, 97, 129, 148, 161
VDU 116, 198
VERIFY 126, 129, 130, 133, 179

X

X-axis 74
X-coordinate 95, 107

Y

Y-axis 74
Y-coordinate 95, 107

Z

Z80 micro processor 153, 155, 166

