

**EVERYTHING YOU ALWAYS WANTED TO KNOW ABOUT
THE SPECTRUM**

The Complete Spectrum assembles the tried and tested work of a number of established authors into a step-by-step guide to mastering the Spectrum. Using this unique single-volume manual, even a complete beginner will soon learn the simple techniques he needs to create his first programs in BASIC, and can quickly move on to more advanced work with graphics, sound, and machine code. It's a reference book, too, with a helpful guide to Spectrum-compatible hardware and software and dozens of invaluable program examples, tables, diagrams and explanations.

The Complete Spectrum is for all who want to use their computer to the full.

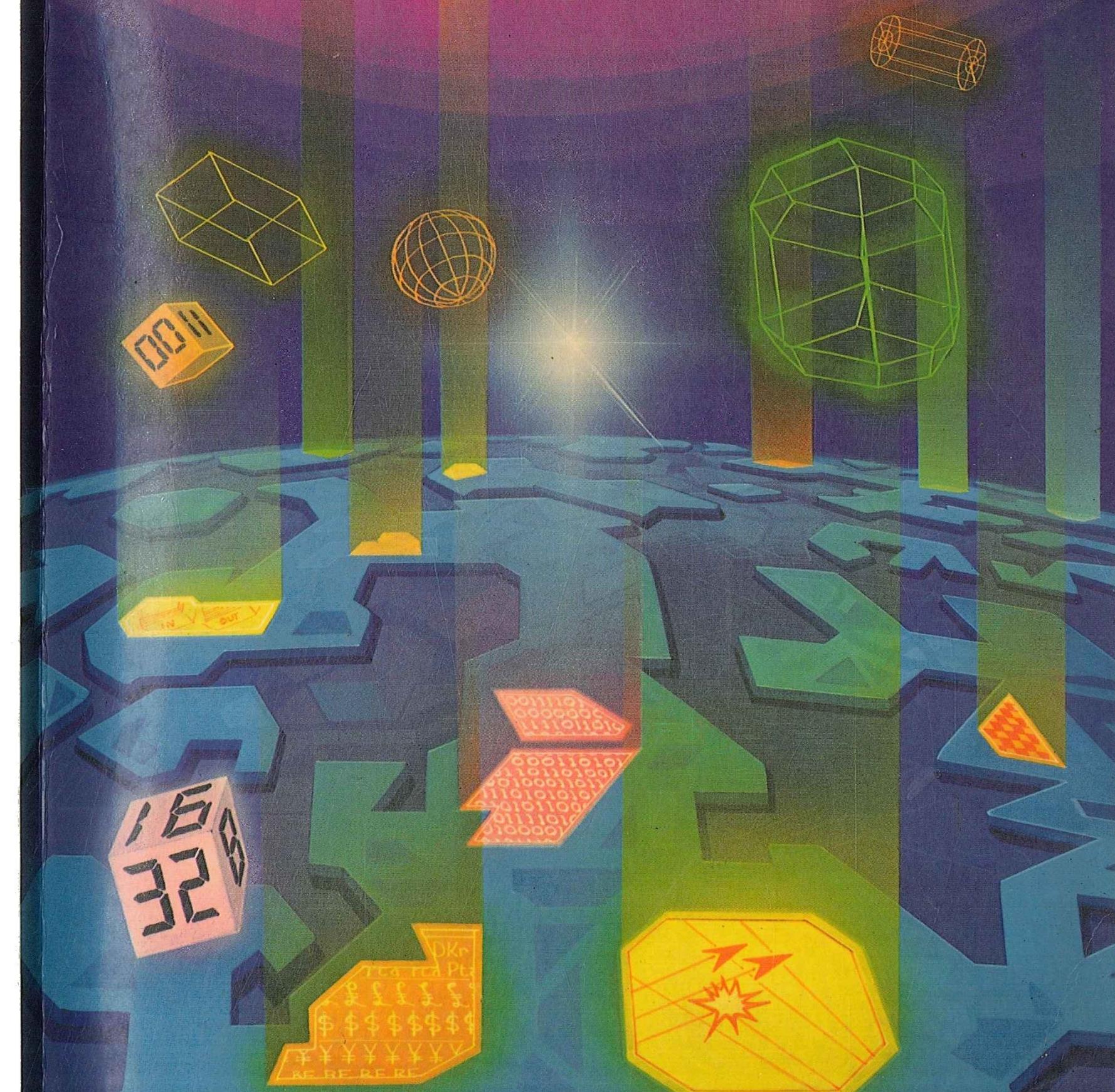
The Editor

Allan Scott is a professional writer and editor and an award-winning audiovisual producer. He uses computers for graphics design, word processing, and data handling, and has built up a complete home system based on the Spectrum.

THE COMPLETE SPECTRUM

EDITED BY ALLAN SCOTT

THE COMPLETE SPECTRUM



EDITED BY ALLAN SCOTT

Front cover illustration by Angus McKie

GRANADA PUBLISHING
Printed in Great Britain
0 246 12569 1

£9.95 net

GRANADA

The Complete Spectrum

**Editor
Allan Scott**

GRANADA
London Toronto Sydney New York

Granada Technical Books
Granada Publishing Ltd
8 Grafton Street, London W1X 3LA

First published in Great Britain by
Granada Publishing 1984

Distributed in the United States of America
by Sheridan House, Inc.

Introduction and selection copyright © Allan Scott 1984

British Library Cataloguing in Publication Data

Scott, Allan

The complete Spectrum.

1. Sinclair ZX Spectrum (Computer)

I. Title

001.64'04 QA76.8.S625

ISBN 0 246-12569-1

Typeset by V & M Graphics Ltd, Aylesbury, Bucks

Printed and bound in Great Britain by
Mackays of Chatham, Kent

All rights reserved. No part of this publication may
be reproduced, stored in a retrieval system, or
transmitted, in any form, or by any means, electronic,
mechanical, photocopying, recording or otherwise,
without the prior permission of the publishers.

Some of the material in this book has previously been published in *The ZX Spectrum and How to Get the Most From It* by Ian Sinclair, *Spectrum Graphics and Sound* by Steve Money, *40 Educational Games for the Spectrum* by Vince Apps, *The Spectrum Book of Games* by Mike James, S. M. Gee and Kay Ewbank, *Introducing Spectrum Machine Code* by Ian Sinclair and *The Spectrum Add-on Guide* by Allan Scott.

Contents

Preface

vii

| | | | |
|---------------|------------------------------------|--|-----|
| PART 1 | GETTING STARTED | by Ian Sinclair | 1 |
| 1 | Setting Up | | 5 |
| 2 | Screen Printing | | 17 |
| 3 | Introducing Variables | | 27 |
| 4 | Loops, Repetitions and Arrays | | 37 |
| 5 | Handling Data | | 51 |
| 6 | Do-it-yourself Programming | | 63 |
| 7 | Other Instructions | | 73 |
| PART 2 | GRAPHICS AND SOUND | by Ian Sinclair and Steve Money | 77 |
| 8 | Screen Displays | | 81 |
| 9 | Getting it in Colour | | 95 |
| 10 | Drawing Techniques | | 101 |
| 11 | Making the Most of High Resolution | | 119 |
| 12 | The Sound of Spectrum | | 129 |
| PART 3 | LEISURE SECTION | by Vince Apps, Mike James, S. M. Gee and Kay Ewbank | 135 |
| 13 | Games for Younger Users | | 139 |
| 1 | Village | | 141 |
| 2 | Multiplication and Division | | 147 |
| 3 | English/French | | 148 |
| 4 | Constellations | | 155 |
| 5 | Spelling Test | | 159 |
| 6 | Areas | | 163 |
| 7 | Submarine | | 167 |
| 8 | Word Search | | 171 |
| 9 | Noughts and Crosses | | 175 |

| | | | | |
|--|---|-----|--|-----|
| 14 | Games for Experienced Users | 179 | PART 7 APPENDICES | 447 |
| 1 | Sheepdog Trials | 181 | <i>Appendix A: Selected List of Spectrum Software</i> | 463 |
| 2 | Laser Attack | 187 | <i>Appendix B: Getting the Most from Mail order</i> | 459 |
| 3 | Spectrum Dice | 195 | <i>Appendix C: A Short Add-on Atlas</i> | 463 |
| 4 | Rainbow Squash | 199 | <i>Appendix D: 697 Z-80 Operating Codes with Mnemonics, Hex Codes, Denary Codes and Binary Codes</i> | 471 |
| 5 | Bobsleigh | 203 | | |
| 6 | Spectrum Smalltalk | 207 | | |
| PART 4 ADVENTURES WITH THE SPECTRUM by Allan Scott | | 217 | <i>Index</i> | 483 |
| 15 | A Sense of Adventure | 219 | | |
| 16 | Telling the Story | 223 | | |
| 17 | Programmed for Adventure | 231 | | |
| 18 | Finishing Touches | 245 | | |
| PART 5 EXTENDING THE SPECTRUM by Allan Scott and Ian Sinclair | | 255 | | |
| 19 | The Spectrum at Work | 261 | | |
| 20 | Add-ons for Games | 275 | | |
| 21 | New Keys for Old | 285 | | |
| 22 | Enlarging the Memory | 291 | | |
| 23 | Cassette Decks, Microdrive and Disk Drive | 297 | | |
| 24 | Speech, Sound and Music | 313 | | |
| 25 | The Computerised Drawing Board | 321 | | |
| 26 | Getting it in Print | 329 | | |
| 27 | Calling all Spectrums ... | 337 | | |
| 28 | TVs and Monitors | 349 | | |
| 29 | The Spectrum in Control | 353 | | |
| PART 6 INTRODUCING MACHINE CODE by Ian Sinclair | | 359 | | |
| 30 | Inside the Case | 363 | | |
| 31 | The Miraculous Microchip | 375 | | |
| 32 | Registers, Addresses, and Assembly Language | 387 | | |
| 33 | What the Registers Do | 399 | | |
| 34 | Programming in Machine Code | 413 | | |
| 35 | Using the Ports | 425 | | |
| 36 | Monitors, Assemblers, and Disassemblers | 437 | | |

Preface

The Complete Spectrum is your guide to a personal journey of computing discovery. From the day you take your new Spectrum out of its box to the day you start thinking about machine code programming, this book will give you the information, advice, and direction you need. Because it covers such a wide range of topics, *The Complete Spectrum* is also the ideal family computer book – there's something here for all ages and all interests, even for those who just want to play games!

If you're more serious about your Spectrum – if you want to develop a thorough knowledge and understanding of your new computer, and its many capabilities – this book has been planned as a step-by-step introduction to serious computing, with a few pleasant but useful excursions into fun and games along the way. *The Complete Spectrum* will help you to write your own programs, but it will also introduce you to some of the better commercial programs on the market, and to some of the plug-in devices – *peripherals* – that extend the range of different jobs your Spectrum can perform. When you've worked through the main text you'll find a great deal of useful information in the Appendices at the back, including a list of selected programs to help you choose from the massive range available for the Spectrum.

Part 1, Getting Started, is ideal for people who have never seen, let alone handled a computer before. It shows you how to set up your Spectrum, connect it to the TV set and cassette recorder, and start writing your very first programs in BASIC – a computer language designed to make life a little easier for human beings.

Part 2, Graphics and Sound, takes things a little further, introducing techniques and ideas to get the most out of the Spectrum's facilities. This requires a small amount of maths, but the information you need is given in a way that even non-mathematicians shouldn't find too hard, and the results can be very satisfying indeed.

Part 3, The Leisure Section, can be just that and no more. It contains full BASIC listings of some selected fun and educational games that you can simply copy into the computer for yourself and enjoy. Younger readers in particular should find this section useful – many of the programs have been written with them in mind. On the other hand, the programs also illustrate many useful BASIC techniques in action, and if you are interested in learning about them then these tried and tested routines should help you understand the practical uses for all the theory you've read about in the first two sections.

Part 4, Adventures with the Spectrum, introduces another fascinating leisure and educational area – adventure games. Many excellent games are now available commercially, but for those who want to write their own this section provides a few basic ideas, and an introduction to the use of commercial software aids – tools that will let you write a complete adventure without really worrying about the details of the programming! However, for those keen to tackle adventuring themselves, or frustrated by the restrictions of the commercial programs, there are also some ideas for putting together simple adventures of your own.

Part 5, Expanding the Spectrum, is an introduction to the enormous range of plug-in and add-on units that can turn your Spectrum into almost anything you want it to be – a superb arcade-level games machine, a word processor and data system, a massive information storage and retrieval system, a sophisticated speech and music synthesiser, an electronic drawing board, a way of turning displays on a TV screen into printing on paper, and a sophisticated communications device that can exchange programs, data and messages with other computers in the same building, or hundreds of miles away at the other end of a telephone line.

Part 6, Introducing Machine Code, crosses what might be described as the ‘final frontier’ of Spectrum computing and introduces the language – *machine code* – that the computer itself uses to process information. By the time you’ve finished this section you’ll be confident enough to tackle some of the more advanced textbooks and manuals that may, at the moment, look like so much double-Dutch.

Finally, don’t skip the Appendices. Here you’ll find a great deal of useful information that couldn’t be fitted sensibly into individual chapters, but will still help you use your Spectrum more easily and more effectively.

Welcome to *The Complete Spectrum*. We hope it will make your personal journey of discovery comfortable, enjoyable, and fulfilling.

Part 1

Getting Started

by Ian Sinclair

Getting Started

If the Sinclair Spectrum is your first computer, then it's a wise choice; if not, it's *still* a wise choice! The Spectrum is an ideal beginner's computer, because it doesn't require any typing skills to use it, it has all the facilities of other computers costing up to twice its price, it's truly versatile, and the computer language called BASIC which it uses is easy to learn and use. However, if you are just starting out you may not have much idea just what it can do and just how it can do it.

If, on the other hand, the Spectrum is not your first computer, and you have come to it from something with less memory space and fewer functions, you will be anxious to get to grips with its more advanced facilities. In that case you'll be glad to hear that this section alone will take you beyond what you can learn from Spectrum's main manual. In fairness, this isn't really difficult – if a manual were going to be a complete guide to everything the computer could do it would probably be published round about the same time as the computer itself was replaced by a new model. Something we hope to show you all through this book is that anyone, anywhere, can always find out something new about their computer, something that even the manufacturers may not always know about! For that reason we don't claim to cover every possible angle – even the section on machine code at the end of the book is only an introduction to the subject. What we *are* aiming to do is to give you what you need to discover Spectrum for yourself. If you're really keen, and want more information and more contact with what other people are doing, subscribe to one of the specialist Spectrum magazines, or join your local user group (to contact them, check out the magazines or your local library).

This section, and many other parts of the book, have been produced by close work with Spectrum itself, and most of the program listings in this section have been printed on the ZX printer (still available, but no longer in actual production, at the time of writing).

Chapter One

Setting Up

If you don't already own a Spectrum, then you will be anxious to know just what's inside the large and exciting-looking box you will have seen in the shops. Surprisingly enough, much of it is packing and accessories – Spectrum itself is actually a bit smaller than this book! (If you remember that everything *in* this book is still only a fraction of what the computer can do, you'll begin to realise what a good investment you've made!) However, as well as the computer itself you should find a mains unit, two sets of cables, two manuals (a thick one with a spiral binding and a thinner one) and a cassette tape called **HORIZONS**.

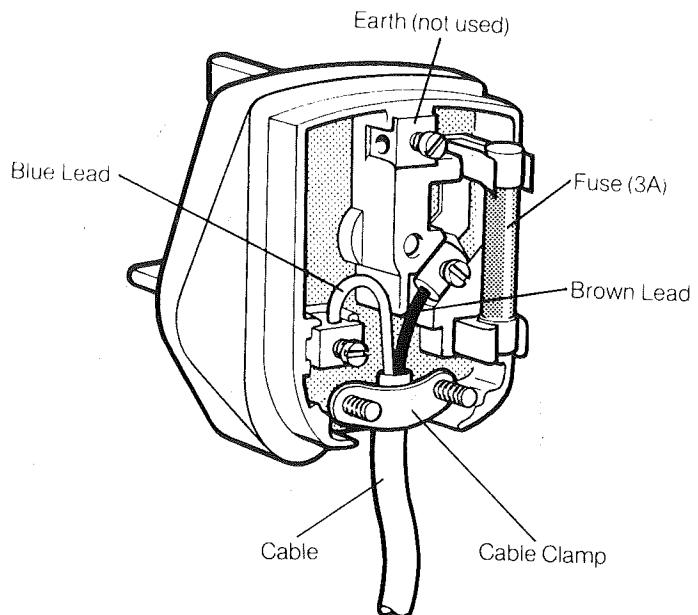


Fig. 1.1. The UK type of 3-pin plug, with connections. The cable clamp should be tight enough to prevent the cable from pulling out, and the wires should be completely clamped by the screws on the brass ends of the pins.

The first thing you'll need to do is fit a mains plug (not supplied), and if this bothers you, take a look at Fig. 1.1. (Like a good many other people, I can't always remember which colour belongs where without checking!) If you're still not sure you can fit it safely, take it down to your local electrician. British plugs should be

fitted with a 3A fuse, or better still a 1A fuse. Readers in the US may find that a plug is already fitted.

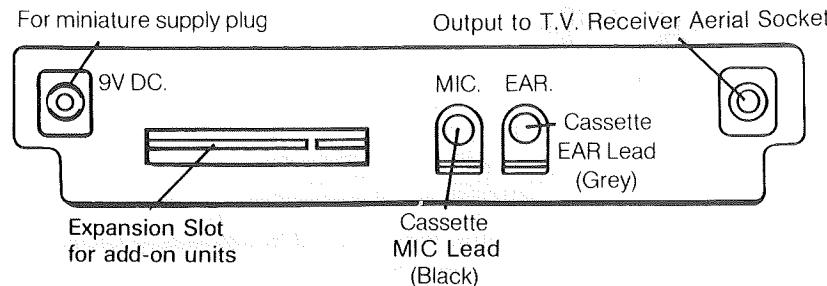


Fig. 1.2. The back of the Spectrum. The sockets are of different types to avoid possible confusion. Be careful not to let metal objects stray into the expansion slot – it's a good idea to tape it over until you use it. You'll find much more about the expansion slot in Part 5 of this book.

To switch on the computer, just plug in the mains unit and then connect the lead from the unit to the power socket at the back of Spectrum (see Fig. 1.2). The manual suggests you use this regularly to turn the computer on and off, but if you're using Spectrum a good deal this isn't really a very good idea – the plug and socket will eventually start to make intermittent contact with each other and you'll find whole programs disappearing without warning as the power supply is cut off. It's better to use the mains plug (which also prevents the power supply unit from becoming overheated) or to fit a switch at some convenient point between the mains plug and the power supply unit. Again, get an electrician to do this for you if you don't feel confident enough to tackle it yourself.

Once the Spectrum is plugged into the mains, the computer has everything it needs to start working. You, on the other hand, need some way of seeing what's going on – in this case, your TV – and some way of storing the work you've done on the computer. This may be a ZX or Dean Alphacom Printer, which will print things out on paper for you, or it may be a cassette recorder, which will allow you to keep both information and actual programs for the Spectrum on tape. It can even be both!

If you turn the back of the computer round to face you, and then take another look at Fig. 1.2, you'll see where everything is supposed to plug in. The printer plugs into the slot on the left – be careful to fit the peg, or 'key', into the matching slot on the printed circuit-board you can see inside the computer (Fig. 1.3). Be gentle with this connection – what you can see inside is the main working circuitry of Spectrum itself, and if you damage it your computer will be useless and you'll have a very expensive bill. Keep metal objects out of this 'user port' and if you have small, inquisitive children, tape it over until you want to plug something into it.

Now for the TV. The Spectrum can produce spectacular colour displays, but only, of course, on a colour TV! Second-hand colour TVs aren't expensive these days, but it's perfectly possible to do your programming on a little black and white TV – one Spectrum owner I know uses a portable that also includes a very

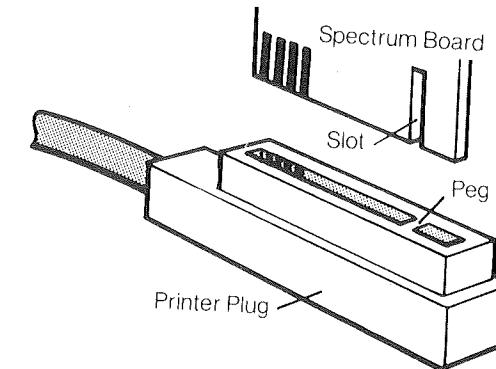


Fig. 1.3. How the printer plug engages with the Spectrum expansion slot.

convenient cassette deck! If you really want colour displays you can always steal a little time on the family TV when no one else is watching it, and in any case you'll find the letters actually look clearer in black and white, which should save you a good deal of eyestrain. However, you can't just haul grandad's ancient TV out of the loft – it must be able to receive modern 625-line signals. In Britain, if it can be tuned to BBC2 it'll be OK, and any colour set should be fine unless you have a very old Spectrum. To connect it up you'll need one of those cables, the one with the plugs shown in Fig. 1.4. To save wear and tear on your patience and the TV aerial socket it's worth buying an aerial splitter like the one in Fig. 1.5 – plug the cable from the Spectrum into one socket, the TV aerial cable into the other, and then fit the whole thing to the back of your TV. Make sure your computer is plugged in, then switch on the TV and turn down the sound. If you really want to see why, then *don't* turn down the sound, but please don't complain to us about the noise!

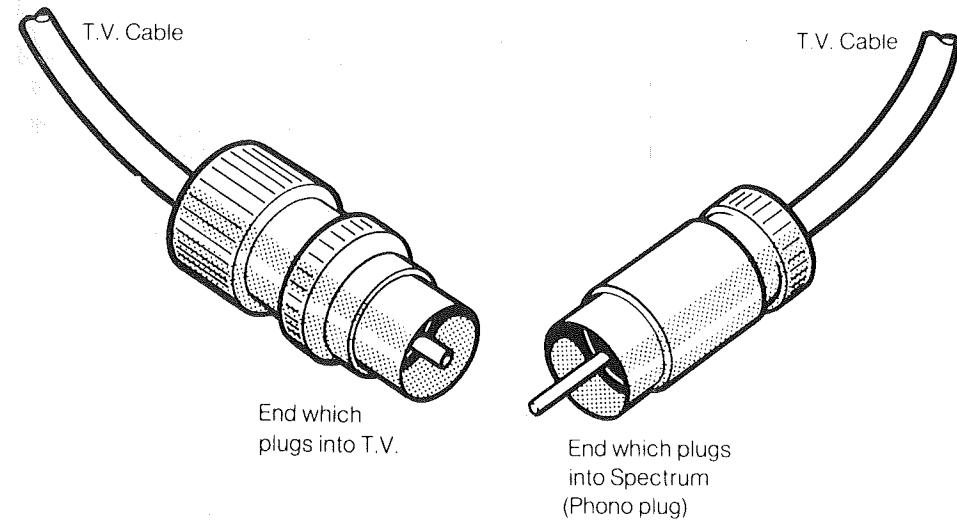


Fig. 1.4. The two different plugs on the TV connector cable.

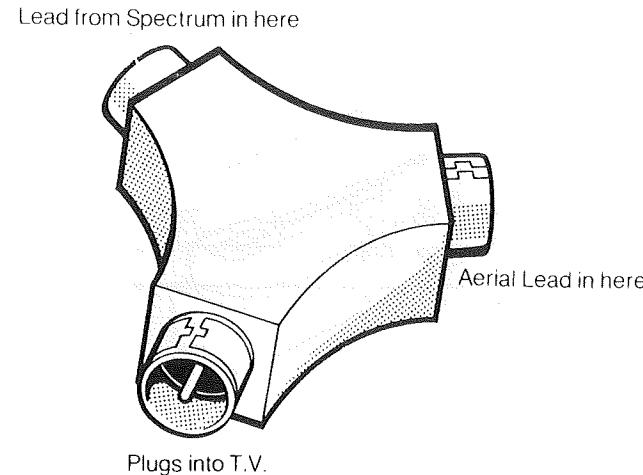


Fig. 1.5. A typical two-into-one adaptor for TV aerial leads. This allows the Spectrum to share the TV with Coronation Street, but not at the same time!

If you don't get a picture at this stage, don't worry. The TV needs to be tuned to receive the Spectrum signal, just as it needs tuning to an ordinary broadcast signal (if you're interested in *all* the details, take a look at Chapter 28). The Spectrum uses Channel 36 – on a black and white portable, just spin the tuning dial. Figure 1.6 shows some of the other possibilities you may come across. Make sure you're using a button or touchpad that isn't normally used for selecting a TV station. Incidentally, the Spectrum frequency is very close indeed to the one used by some video recorders, so you may find that the Spectrum will interfere with signals from this source. It's not a good idea to have both machines switched on at once.

When the tuning is correct you should see a white screen with the words

© 1983 Sinclair Research Ltd

in black near the bottom of the screen. Take time over the adjustments. If you try to make them too quickly you may go straight past the signal you're looking for. When you do get the lettering, it's time for some fine adjustment. On a black and white set, adjust the contrast and brightness controls until the letters are clear black on white, and the white isn't blinding you. Only one position will give you a really good picture, and Fig. 1.7 shows the faults you're trying to avoid. Tuning a colour set is trickier – it's easier if the Spectrum is producing a colour, so press the key marked BORDER (6th along on the bottom row), then the one marked 6, then the one marked ENTER. The result should be a yellow 'frame' all around the screen with the words 'OK. 0:1' at the bottom left. Adjust the set till you get a clear, bright yellow with reasonably clear letters (though they're likely to be a little fuzzier than they would be on a black and white set). Again, if your colour TV has a contrast control it may be worth turning it down a little, if it hasn't, then the adjustment will need to be made by an experienced TV engineer.

If your TV uses preset pushbuttons or touch controls, or even a remote control, it's worth leaving one 'station' permanently tuned for your Spectrum – otherwise

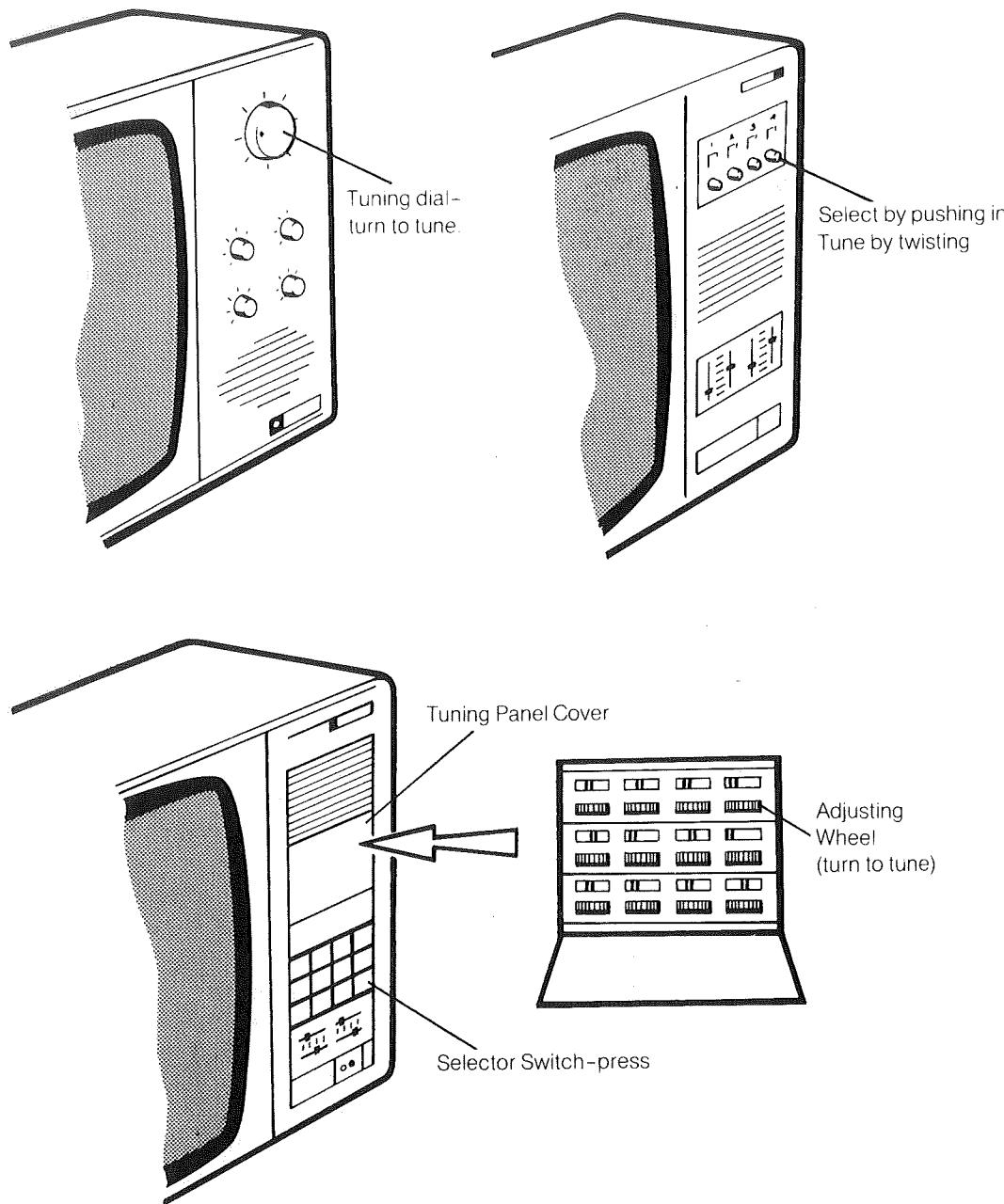


Fig. 1.6. TV tuning controls. (a) Single dial, as used on B&W portables, (b) four-button type, (c) latest 12-switch type with tuning panel.

make a note of the control settings. If you find black letters on a white screen too much of a strain on the eyes, then try this:



'Ghost' images
after a letter



Fuzzy letter shape



White 'blobs' between letters

Fig. 1.7. Tuning faults and how they show up.

- (1) Press the BORDER key, then the 0 key, then ENTER.
- (2) Press the CAPS SHIFT and SYMBOL SHIFT keys together, then hold down the SYMBOL SHIFT key and press the key marked C with the word underneath it. Then press the 7 key and ENTER. Finally, press CLS and ENTER.
- (3) Press the CAPS SHIFT and SYMBOL SHIFT keys together, then hold down the SYMBOL SHIFT key and press the key marked X with INK in red underneath it. Then press the 7 key and ENTER. Finally, press CLS and ENTER.

Now that's done you should have a completely black screen with white lettering, and, incidentally, a good test of your tuning. The letters should still be perfectly clear – if they're not, try retuning.

Getting things taped

There are two sockets at the back of the Spectrum that have still not been used – the two for your cassette recorder. If you're new to computing these may puzzle you. Why, after all, do you need to tape record a computer?

A computer doesn't do anything it isn't told to do. You've already entered some *immediate commands* to change INK and PAPER and BORDER colours, but once the computer has obeyed these commands, it forgets about them. Computer *programs* are different – they are sequences of commands which are held in the computer's memory, and carried out when you enter a command to begin the sequence. Many modern washing machines use small computers called *microprocessors* that carry permanent, built-in programs. The result is that you can just put in the clothes and press a button, instead of filling the tub with hot water, emptying it, dropping the soap in at the right time, and all the hundred and one other things you'd have to do with a simpler machine.

One particularly useful Spectrum feature is that command words can

be entered without typing them out. As you can see if you look at the keyboard, most of them can be entered just by pressing one key. For a long program, though, this can still take you quite a lot of time and effort, and if you switch off, or use the NEW command, everything you've typed will be lost. What you need is a way of keeping that work so that it can be fed easily back into the computer another time without typing it all out again. That's where the cassette recorder comes in. You don't actually record the program, just a set of electrical signals that will produce the program again when they're played back into the Spectrum. So before you start learning to program the Spectrum you need to know how to 'load' commercial programs from cassettes (especially the HORIZONS tape), how to record (or SAVE) your own programs on cassette, and how to check that the recording is accurate by using the VERIFY command.

Start by plugging the double lead into the remaining sockets at the back of the computer, with the grey plug in the socket marked EAR and the black one in the socket marked MIC. (More about this in a moment.) The cassette recorder you use doesn't have to be the latest in hi-fi stereo reproduction, and it's usually better if it isn't. What you want is a straightforward portable cassette recorder which will work on mains as well as on battery power. I have found the Trophy CR100 from Currys to be excellent, because it has a tape counter you can use to check the position of each program on your tapes. Provided that your recorder has a microphone (MIC) socket and an earphone (EAR) socket, it should be suitable; and for your first test try playing the HORIZONS cassette into your computer. This is called 'loading'.

Start by plugging the MIC plug into the MIC socket of the cassette recorder. If you followed my advice a few pages back, this will be the black plug – in any case it must be the one that leads to the socket marked MIC on the Spectrum. Now connect the other (grey) plug to the socket marked EAR and put the HORIZONS cassette into your cassette recorder. Don't try to play the tape yet. Now press the key marked LOAD on the Spectrum keyboard, then hold down the key marked SYMBOL SHIFT and press the P key twice. This will produce the message LOAD "" on your TV screen. Press the ENTER key and then start your tape by pressing the PLAY key or button on your cassette recorder.

You should now see a rather entertaining display of modern art on your TV set – first a set of moving red and blue stripes, then a pause, then moving black and yellow stripes. If you don't get these, or they stop with a screen message saying 'R Tape loading error 0:1' then rewind your tape, hold down the CAPS SHIFT key, press the BREAK key, and try adjusting the volume control on your cassette deck (usually it needs to be louder). Keep trying until the TV screen shows the message 'VOLUME SET CORRECTLY PLEASE WAIT'. This doesn't happen with every program you buy, so once the volume is right, leave it alone or make a note of the setting. After this, follow the instructions on the HORIZONS tape and you will find out a good deal about your computer. However, you may find it easier to use the tape in conjunction with this book!

You now know how to load a commercial tape, but suppose you want to load one of your own?

I'll assume that you've just switched on your Spectrum, and you have the copyright notice displayed. (If you haven't, then take the plug out at the mains and put it back in again.) Press the ENTER key, and you will see a flashing letter K near the bottom left-hand side of the screen. This is called the 'cursor', it marks where the next typed letter will be placed on the screen, and it also reminds you what you are doing. When it has the shape of a letter K, it is there to remind you that you are about to enter a **KEYWORD** – one of the instructions that Spectrum can recognise and act upon. We'll use a simple set of instructions – press 1 (top left) followed by REM (2nd row from top, 3rd along). Now press ENTER (3rd row, right hand side) and you should see appearing at the top of the screen:

1>REM

and the letter K should again be winking at you near the bottom left-hand side. This time, press 2, then REM, and ENTER, so that the top of the screen shows:

1 REM
2>REM

The > reminder is there to show you that the second line is the 'current line', the one you have been working on most recently. Keep on in this way, using a different number each time, until you have five lines. You'll notice that if you enter them out of order (like 1, 2, 5, 4, 3), they will be arranged in the correct order at the top of the screen. After the 5th line has been entered, you will still have the K winking at you.

This, believe it or not, is a program. It won't do anything, because the instruction REM is just to let the computer know that what follows is a reminder rather than something to be done, but it is enough to test the cassette operation. With the K visible, press the key marked SAVE (on the S key, 3rd row, 2nd left), and you will see the word SAVE appear on the screen, with the flashing cursor changed to an L. Now press the SYMBOL SHIFT key and while you hold it down, press the P key briefly. Don't hold the P key down, or else it may repeat. You will see printed in red on the P key the inverted commas, or quote marks, ("), and that's what should appear on the screen. Release the SYMBOL SHIFT key and press T – which will give you a 't' on the screen following the ". Now press the SYMBOL SHIFT key again with the P key to get another quote mark in place so that the line reads:

SAVE"t" L

with the L still flashing. Press the ENTER key, and the screen will clear and display the message:

Start tape, then press any key

Do just that. Make sure that you have a cassette in the recorder, and that it has been wound on from the start so that there is some tape available, *not* the plastic ribbon which is used at the start and end of each cassette. Press the RECORD and PLAY keys of the recorder together, so that they both stay down, and when you are sure that the motor of the recorder is running, press *any* key on the Spectrum.

You will then get some more interesting displays on the screen. Enjoy them, but don't switch off when the screen goes blank at first, wait until you see the message on the bottom line, which will read:

OK, Ø:1

Now press the STOP key of your recorder, and your first 'program' should be safely recorded.

You can't feel it, taste it or smell it, so how do you know it's there? Easy – Spectrum, as you might expect of an advanced design of computer, can compare what is on the tape with what ought to be there. The program, you see, is still stored in the memory of the Spectrum, and if we play back the tape, Spectrum can check to see that the recording is a good one. This is done using the VERIFY command. Here goes.

Wind back the tape. It doesn't matter if you wind back too far, Spectrum will sort that out. Play the tape with the EAR plug out, ignoring Spectrum for the moment, just to make sure that you have something recorded. It's easy to find your place if you have a recorder with a tape counter, but you may have some trouble finding the start of your program otherwise unless you have used the start of a short tape for your recording. If you hear some unpleasant sounds that suggest a modern composer on an off-day, that's your program. The noise should be uncomfortably loud with the volume control half-way up. If your recorder has a TONE control, put it in the position that gives maximum treble, the most shrill sound. Now stop the playback, and wind back to the start again.

Put in the EAR plug. Going back to the Spectrum, hold down the SYMBOL SHIFT, press and release the CAPS SHIFT, and then press R (2nd row, 4th from left) and release the keys. This should give you the word VERIFY on the bottom of the screen. Now, keeping the SYMBOL SHIFT pressed, press the P key to get ", then release SYMBOL SHIFT and press the T key. Press SYMBOL SHIFT again, then P to get another ", so that the bottom line reads:

VERIFY "t"

and then press ENTER.

Not a lot seems to happen until you press the PLAY key of the cassette recorder. With the volume control of the recorder set at half-way, you should hear nothing (the EAR plug is inserted, I hope) but you will see the same display as you got when you recorded the program, with

PROGRAM t

appearing on the screen at first, followed by

OK Ø:1

finally, when the display has cleared. If this is what you see, congratulations, you have found the correct settings, and you can STOP the recorder.

If you don't get these messages, then stop the recorder, leaving the Spectrum as it is. Rewind the tape (easy with a counter!) and try again with a different volume

control setting – you only need to run the tape, don't touch Spectrum. Keep trying until you find a volume control setting at which the stripes appear, then the messages. If the tape has recorded correctly there will be a suitable volume control setting *somewhere*. When you find it, mark the setting – you don't want to have to go over all that again, do you? If you simply can't find it, then press the CAPS SHIFT and SPACE keys together (the Break action), and take a look at the check list of Fig. 1.8, and make a new recording. Don't just give up, because this is something you just have to master.

Error Check List

1. Play the tape back with all plugs out, and listen to it. Does it sound like a shrill rasping sound? If it doesn't, or if all you can hear is a low-pitched hum, then the program is not recorded.
 - (a) Check that you have used the same colour plugs for the MIC connection at each end (Spectrum and recorder).
 - (b) Check that your recorder works by recording a message on it using a microphone, and replaying this.
 - (c) Try again. If there is no recording you may possibly have a faulty Spectrum.
2. If the recording exists but sounds faint, then you may have made the recording with the EAR plug in place. Try again, making sure that this plug is out when you record.
3. If the recording sounds loud when you play it back, then it ought to operate the Spectrum.
 - (a) Check that both ends of the replay line are connected when you replay with the same colour of plug in the EAR sockets at each end.
 - (b) Try the VERIFY routine several times at various volume settings.
 - (c) Try again. If the recording refuses to VERIFY there may be a fault in your Spectrum.

IN GENERAL, faults are more likely to be due to the recorder than the Spectrum.

Do not use C120 tapes. C90 is the longest tape length which is reliable. Use well-known tape brands, like AGFA, BASF, Maxell, Scotch, Sony, etc., or the short tapes (C5, C10, C15) which can be obtained from computer stores.

Fig. 1.8. A check list to go through if you have cassette loading problems.

Appendix A lists a selection of the many programs available ready-recorded on cassettes. Sometimes these will not load at any setting of the volume control, or will need an unusually high volume control setting. This is usually caused by a recorder problem – a recorder head which has been assembled slightly squint. You can sort this out for yourself – see Fig. 23.1. Still on the subject, you can also find that after a lot of use, your programs don't

seem to load (replay) so reliably. This is a sign of a dirty recording head or pinchwheel (the wheel that pulls the tape along) on the recorder. To sort it out, buy a cassette head-cleaning kit from any audio store, and follow the instructions.

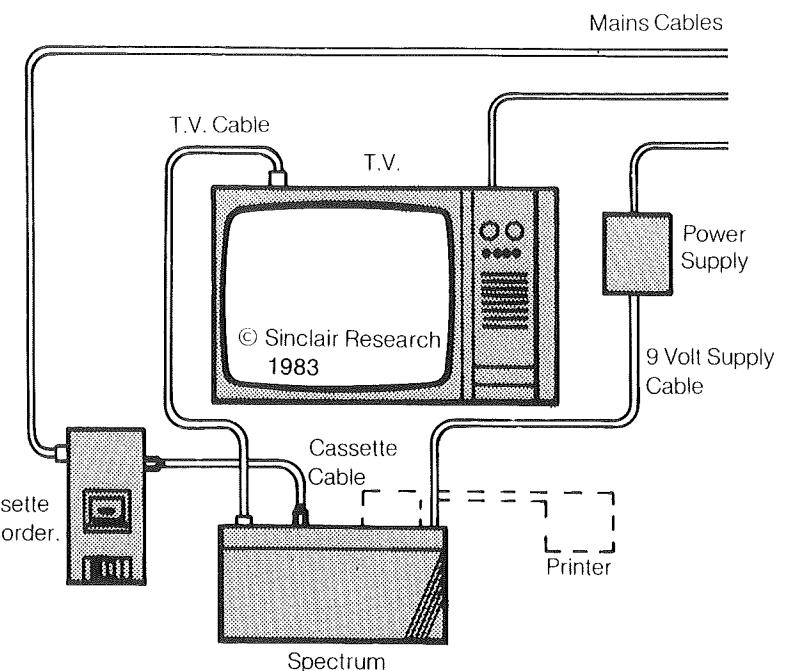


Fig. 1.9. Connected-up components of the Spectrum system.

The keyboard

By this time, you will be starting to see how the keyboard operates. Each key carries several symbols or words, and there are others printed above and below the keys. It all looks very baffling at first, but as you get to know your Spectrum, you'll find it remarkably easy to use. What follows is a brief summary – we'll remind you as we go through the chapters.

When the K cursor is flashing on the bottom line, what the computer expects you to type is either a line number or a Keyword of instruction, and if you press any of the letter keys, you will get the *word* that is written on the key. For example, pressing Q gives PLOT, pressing P gives PRINT, pressing K gives LIST and so on. One key enters a complete word. If you press a number key to start with, you simply get the number, which the computer will use as a line number. Each program line in the BASIC computer 'language' starts with a line number and then a keyword, so that quite a fair chunk of the program line is put into memory with just a few keys. Between instruction Keywords, however, you need to be able to enter numbers and letters which you might, for

example, want to appear on the screen during the program. When the computer is ready to accept these, the K cursor changes to a flashing L (L for Letter), so that when you press key Q you get q, when you press key G you get g, and so on. You will get the small letters, called 'lower case' when you do this unless you hold down the CAPS SHIFT key at the same time. If you want to type a message in capitals (upper case), you can lock the CAPS SHIFT by pressing CAPS SHIFT and the 2 key at the same time. This operates the CAPS LOCK (written in above the 2), and ensures that all letters appear in capitals until you repeat the operation of pressing CAPS SHIFT and 2 together.

Now you will also find more words and most of the important signs like =, ., *, ; written in red on the keys. You can get these when the cursor is showing K or L by holding down the SYMBOL SHIFT key before you press the key you want, and holding down the SYMBOL SHIFT key until you have released the key you want to use. The SYMBOL SHIFT key is lettered in red to remind you of this. If you are sloppy, and you hit the other key just before you press the SYMBOL SHIFT, then you won't get what you want, and you will have to press CAPS SHIFT and Ø (DELETE) together to rub out the mistake.

Now for the words above and below the keys. The words printed in green above the keys are obtained by pressing CAPS SHIFT and SYMBOL SHIFT together, releasing (the cursor now becomes an E) and then pressing the key you want to use. The words *below* the keys are obtained by holding down the SYMBOL SHIFT, pressing CAPS SHIFT and releasing it, then pressing the key you want to use.

The top line of keys is different. The keys give numbers, normally the symbols in red when the SYMBOL SHIFT key is used, and the words or actions shown in white above the keys when CAPS SHIFT is pressed along with the top-line key. The names of colours that are written above some of the keys can be ignored at the moment, and we'll also ignore the shapes (graphics symbols) for the moment. There will be a lot more about these later in Part 2.

For editing, when you are ready, see Chapter 7.

Chapter Two

Screen Printing

Learning to program a computer is rather like learning a foreign language. You need to know some words (vocabulary), and you need to know some rules (syntax). Computers like the Spectrum use one computing language called BASIC (there are many others) which has a 'vocabulary' of about 90 'words' and a few rules of syntax, making it simple to learn. The big difference between trying a program on the Spectrum and trying your school French in Calais is that the computer will understand only *perfect* BASIC – with each word spelled perfectly and used with the correct order. The 'single-key' system that is used for the vocabulary of the Spectrum removes the problem of correct spelling of the instruction words. All you have to do to enter one of these words of instruction (printed in white on the keys) is to press the key which shows the word at a time when the letter K shows a cursor on the bottom line. These words are the ones that you will want to use most frequently; other instruction words are printed in red or in green, and they need the SHIFT keys to be pressed as well as a letter key. We mentioned these in Chapter 1, and we'll deal with the details as we go on. Be careful, incidentally, not to hold a letter key down too long, because the key action will repeat if you hold a key down – useful if you want a lot of or *****, but not when you are typing mmmmyyyyynnammmeeee!

Direct commands

A computer can be used to carry out an instruction directly, without a program. If you type a complete command, and then press the ENTER key (3rd row down, right-hand side), the command will be obeyed. If you want to do it again, you will have to type the whole command again – and that's the difference between a direct command and one which is programmed. Once you have put your command instructions into the form of a program, they will be stored in the memory of the computer, and will be carried out each time you type the special make-it-go instruction, RUN, followed by pressing the ENTER key.

Try this one out. With the computer switched on, the screen clear, and the K cursor showing on the bottom line (switch off and on again if you are not sure how to get to this state), press the key marked PRINT (2nd row down, right-hand side). You will see the word PRINT appear on the bottom line, and the cursor will change to a letter L in place of K. This means that the computer now expects you to enter what you want printed – letters. This is where syntax comes in, though. You can't just type in the letters you want to print, you have to start with a quote

mark (inverted commas, ”). These are shown *in red* on the P key that you pressed to get PRINT, and you can make them appear on the screen by pressing the SYMBOL SHIFT key before and while you press the P key. The idea, remember, is that the words and symbols shown in red on the keys are obtained when you press the SYMBOL SHIFT along with the key. If you get the order wrong, and press the P key *before* you press the SYMBOL SHIFT, you will see the letter p appear. Don't panic – press the CAPS SHIFT and DELETE (on the Ø key) together, getting the CAPS SHIFT key down first, and the p will disappear. If you hold the Ø key down too long, everything will disappear and you will have to start again – remember how these keys repeat.

Once you have the quote mark in place, you can type in something that you want printed just as if you were using a typewriter. If you can't think of anything to type, try Sinclair rules, O.K.? Pressing a letter key, with no other key pressed, while the L cursor is flashing, will give a small letter, and you need to press CAPS SHIFT along with the letter to get capitals. If you want a whole message in capitals, like a title, you can press CAPS SHIFT and CAPS LOCK together, and release both. After this, all letter keys will give capitals. To get back to lower case letters, just press CAPS SHIFT and CAPS LOCK again.

More syntax now. When you have typed your message, you have to indicate to the computer where the message has ended by putting in another quote mark (SYMBOL SHIFT and P). Once you have done that, you can carry out your command by pressing the ENTER key. The message appears on the screen, but not the quote marks, because these were only instructions to the computer to tell it where the start and end of the message were. That's it. If you want to do it again, you have to type the whole thing again. Direct commands can be useful, particularly when you want to discover what the correct syntax of a command is, or for experiments, but most of the time we prefer to put our instructions into the form of a *program*.

Program instructions

Computers don't think for themselves, not even your Spectrum, and they can't easily be told by speaking to them (but it is possible, as we'll see in Part 5!). When you want the computer to carry out a set of instructions that are arranged in sequence (a program, in fact) you need some way of signalling to the computer that this is what you want. In the BASIC language, this is done by starting each instruction or group of instructions with a *line number*. A line number is, for Spectrum, a positive whole number (mathematicians call them *integers*, and since that's shorter, we'll use the word from now on) which should lie between 1 and 9999. The Spectrum will NOT accept a line number greater than 9999, unlike some other computers. For convenience, we usually start our line numbering with 10, and go up in tens, so that a sequence of line numbers might read: 10 20 30 40 50... You don't *have* to do this, but it has the convenient advantage that if you want to place a new instruction between two others, you will have several spare numbers available. If you type the same line number again, and use it, the line which

previously had that number will be rubbed out, but if you have numbered in tens, then you can get nine other lines between any pair in your program, which allows for a lot of 'second thoughts'. Even when you type the lines out of sequence, like 10 20 100 30 60 40, the computer will arrange them in the correct order each time you press the ENTER key – and arrangement in order is the reason for using line numbers. Languages which don't use line numbers are not so easy to use!

For the moment, we'll continue to use the PRINT instruction, because it's the one which causes things to appear on the screen. Without it, nothing appears, so you can't tell whether the computer has carried out your instructions or not. To use the PRINT as a program instruction, then, we need to start with a line number. The computer will be ready to do this when the K cursor is flashing.

Type this line:

1Ø PRINT "Sum"

Remember the drill? With the K cursor flashing, type the digits 1 and then Ø to get the line number. The zero on the keyboard has a slash through it so that you won't mistake it for the letter O – be careful about this, because computers expect you to be the boss and to issue the correct instructions. When the number 1Ø is visible on the bottom line, press the P key to get PRINT, SYMBOL SHIFT and P to get ", then type the word Sum (separate letters, because this is not an instruction). End with the final quote mark, and when you are sure all is well

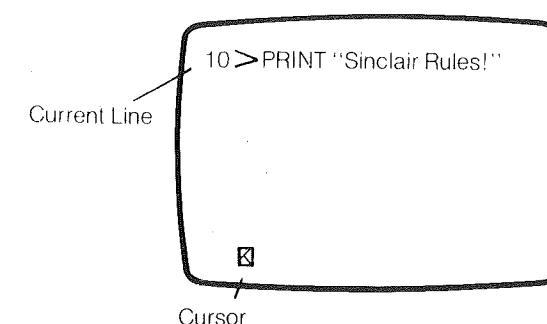
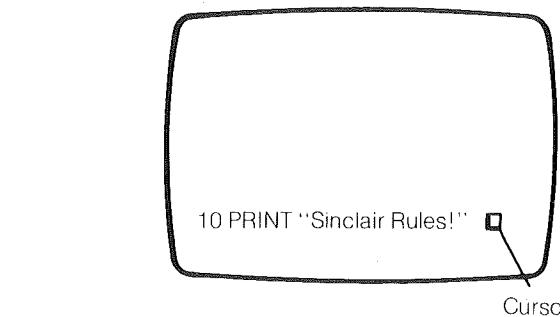


Fig. 2.1. The appearance of the screen as you type a line and after pressing ENTER.

(CAPS SHIFT and DELETE will wipe it out if it is wrong) press ENTER and watch what happens. If your line was typed correctly, it appears on the main part of the screen, like the example in Fig. 2.1, and the bottom line is cleared, leaving the K cursor flashing again. If your line contained an error (not between the quote marks), it will *not* be shifted from the bottom line – a question mark will flash where the error is, and will stay until you delete the error and correct the line. You won't get away with syntax errors when you use Spectrum, but if you typed Sim instead of Sum, the computer takes no notice because it's not a command or instruction, just something you want typed, and you know best!

This is not much of a program, but it will illustrate what programming is all about. Once you have typed it, and pressed the ENTER key (entered the line), the information is stored in the usable memory of the computer, called Random Access Memory, or RAM for short. To make the computer carry out the instructions that are contained in your program you need to use a direct command – RUN. It's easy enough – press the RUN key (on the R key), then press ENTER, and look at the result of your program on the screen. It has printed what you asked, just as you typed it.

If you want to remind yourself of what you asked the computer to do, just press the LIST key (K key), and then ENTER. This command causes the computer to show a list of the lines you typed, in order. If there are more of them than the screen can hold, the list will fill the screen, and you will get the message 'scroll?' on the bottom line of the screen. This means, do you want to see more? If you need to inspect more of the listing, as it's called, press Y for YES, and the listing will continue. If you want time to examine what is on the screen, just ignore the message, and if you want to end the listing at that point, type N (for NO).

With only one line of program, we're not greatly concerned about long lists at the moment, but we can add to our program. With your line 10 still in the memory, and the K cursor still flashing, type in:

```
20 PRINT 2 + 2
30 PRINT 2 * 3
```

These are arithmetic instructions, so there are *no quotes*. We don't want the computer to print $2 + 2$, but the *result* of $2 + 2$ (still 4, even with New Maths), so that quotes are not needed. Without quotes, the computer assumes that you want a result printed, not a set of letters. The + sign is on the K key (using SYMBOL SHIFT), and the star sign which is used for multiplication (x is a letter) is on the B key, using SYMBOL SHIFT again.

Now RUN your program, look at the result, and LIST it. Now you know how to print a message and do a bit of arithmetic, so try this one:

```
10 CLS
20 PRINT "2.76 times 4.15 = ";2.76*4.15
30 PRINT "6.74 divided by 1.82 = ";6.74/1.82
40 PRINT "4.42 squared = ";4.42*4.42
```

CLS means clear the screen, and is on the V key. The divide symbol is / and is

also on the V key, but needs the SYMBOL SHIFT as well. The method of squaring a number is 12, with the up-arrow sign on the H key using SYMBOL SHIFT – for cubes you would use 3 in place of 2. Remember that you have to ENTER each line into memory when it is completed by pressing the ENTER key. Now RUN (remember to ENTER) and LIST (ENTER) your program.

This illustrates rather neatly the difference between printing literally (what is commanded by placing between quotes) and printing a result (no quotes). It also illustrates how much more neatly a computer can do work of this type, as compared to a pocket calculator. It also looks more like a real program, and this gives you another chance to try out the SAVE, VERIFY and LOAD commands. Remind yourself of what is needed by looking at Chapter 1 again, and when you are sure, by using VERIFY, that you have a good recording, turn off the computer and switch it on again. Your program has now been wiped from the memory (try LIST), but it should be stored on the cassette. Wind back the cassette, put the EAR plug in, and press the LOAD key (key J) on the Spectrum. Now put in a quote mark (SYMBOL SHIFT and P), and if you can remember (didn't you make a note of it?) the 'filename' of your recording (what you typed between the quotes when you SAVED it), type the filename and then another quote mark. If you have forgotten, don't worry, just press the quote key again so that your command appears as LOAD"". Press the ENTER key, and then press the PLAY key of the recorder. You should see some interesting patterns on the TV screen, particularly if you are using a colour TV with the colour control turned up, then the name of your program (filename) will appear on the top left-hand side of the picture area. Some more patterns then appear around the picture (the border), then you will get a message on the bottom line to indicate that your program has loaded and that you can now stop the recorder. Don't let the recorder play on if there is any other program on the tape. Test that your program has loaded correctly by using LIST, and then try RUN – then take the EAR plug out again in case you forget it.

Neater printing

So far, we've allowed the computer to decide where it will print everything. Unless instructed otherwise, the computer will take a new line each time you use the PRINT instruction, and will start the line on the left-hand side of the screen. We can alter the position of anything we print by using the extra words TAB and AT. Try the little program of Fig. 2.2 now – and be careful about the semicolons and commas. When you run this one, the word 'example' is placed at the centre of the screen on the first available line. TAB is short for Tabulation, a word that is used by typists to mean the number of spaces along from the start of a line. Tabulation numbers for the Spectrum start with 0 and go to 31. If you use numbers greater than 31, then the actual number that will be used is the *remainder* when the number is divided by 32. If, for example, you type TAB 40, then what you get is TAB 8 – the remainder when 40 is divided by 32 (32 into 40 goes once with 8 remaining, yes?).

PRINT AT uses numbers to mean lines and columns. The first number after

```

10 PRINT TAB 12;"example"
20 PRINT AT 12,9;"2.5+3.7= ";2
.5+3.7
30 PRINT AT 7,11;"ADDITION"
example

```

ADDITION

2.5+3.7= 6.2

Fig. 2.2. A program which uses TAB and AT to control the position of printing.

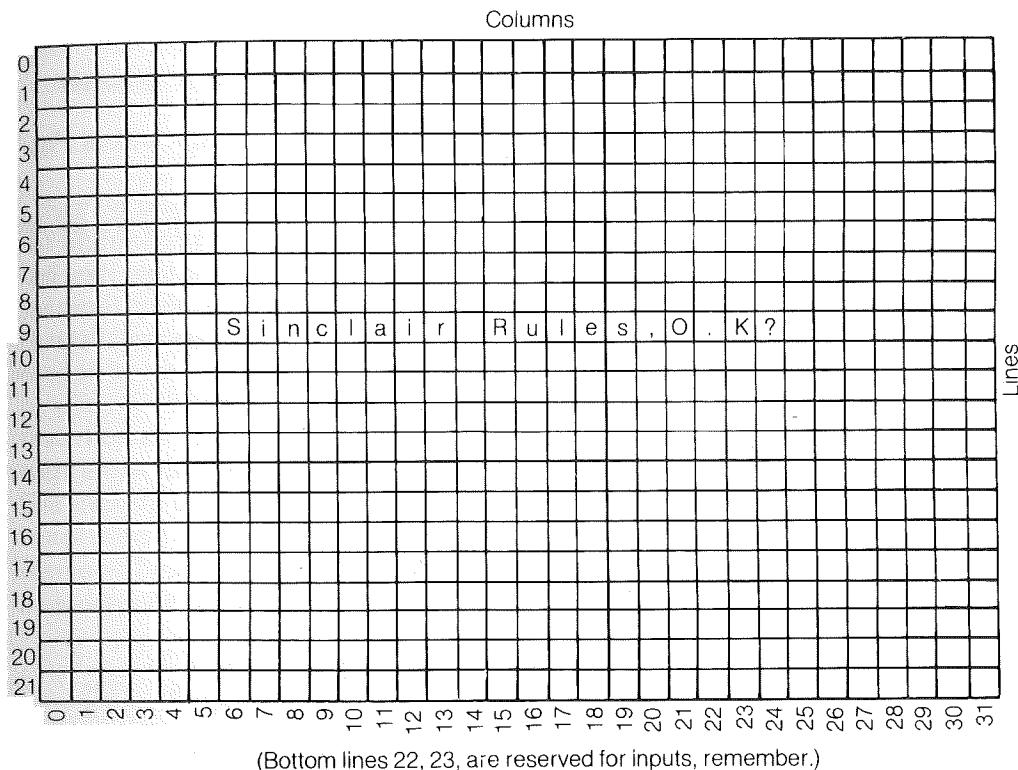
AT (AT is on the I key) is the number of lines *down* the screen from the top, starting with 0 and ending with 21. The two bottom lines are reserved for inputs, and you can't normally PRINT on to them. The second number after AT, separated from the first one by a comma, is the column number, and it's the same as the number you would use in a TAB instruction. Using PRINT AT allows you to print anywhere on the main screen, even if it means wiping out something else, and the printing does not have to be in order – you can print at the bottom before you print at the top, for example. This is illustrated in Fig. 2.2. Figure 2.3 shows how to calculate the TAB number to get a title word centred on the screen, and Fig. 2.4 shows how the screen TAB and AT spaces are numbered.

With the aid of TAB and AT, we can now use the whole area of the screen, apart from the two bottom lines, as we wish. There are other 'print modifiers', as they are called, the symbols ; (semicolon) and , (comma). The semicolon prevents the action of taking a new line, so that you can assemble a line from several PRINT instructions. Take a look at the example in Fig. 2.5. This carries out another piece of simple arithmetic in three separate PRINT instructions, but the use of semicolons in lines 10 and 20 makes sure that each part is printed on the *same* line. This can be very useful when we have two separate PRINT instructions but need both parts in the same line. In the example of Fig. 2.5, we could have used other methods, but in many examples, the use of the semicolon can ensure neater printing.

The comma has quite a different effect. Take a look at the program in Fig. 2.6, which takes us into the realms of 5-line programs. The £ sign is on the X key (using SYMBOL SHIFT). Make quite sure that the commas in lines 10 and 40 are *between* the sets of quotes, because they won't have the same effect otherwise. The effect of commas used in this way is to divide the screen in half, with the first piece of printing on the first half and the second starting in the middle. This does not keep printing on the same half, however, it only controls where printing starts on *that line*. If you have anything to be printed which is of more than 15 characters

Spectrum Independent
20 characters
(count space between m and l)
32 characters per line
32 – 20 = 12
½ of 12 = 6
so use TAB 6
PRINT TAB 6; "SPECTRUM INDEPENDENT"

Fig. 2.3. Finding the correct TAB number to centre a title.



(Bottom lines 22, 23, are reserved for inputs, remember.)

The example shows the result of the command
PRINT AT 9,6; "Sinclair Rules, O.K?"

Fig. 2.4. The numbering of TAB and AT on the screen.

```

10 PRINT "5.6-3.4"
20 PRINT "=";
30 PRINT 5.6-3.4

```

Fig. 2.5. Using semicolons to force printing to stay in one line.

```

10 PRINT "My score","Your score"
20 PRINT 10,5
30 PRINT
40 PRINT "My price","Their price"
50 PRINT "£125","£350+"

```

| | |
|----------|-------------|
| My score | YOUR SCORE |
| 10 | 5 |
| My price | Their price |
| £125 | £350+ |

Fig. 2.6. How a comma can be used to divide printing into two halves of the screen.

(including spaces) long, your neat division of the screen will be lost – look at Fig. 2.7 for example.

There is another way of using the PRINT instruction. PRINT used by itself, with nothing after it, will select a new line, and this was used in these programs to cause the printing to skip a line. Spectrum has another way of doing this, by making use of the apostrophe mark (') which is, in red, on the 7 key. Each time the apostrophe occurs outside quotes, it will cause a new line to be selected. Take a

```

10 PRINT "Try this out-it's too
long"
20 PRINT
30 PRINT "O.K. here-", "-but this
side is too long"

```

| | |
|-----------------------|-------------------|
| Try this out-it's too | long |
| O.K. here- | -but this side is |
| s too long | |

Fig. 2.7. Limitations of the comma – the printed phrases must be short enough to fit into half the screen.

look at the program of Fig. 2.8. Line 10 contains two items to be printed, but the three apostrophes between the PRINT items cause a couple of lines to be skipped and left blank. In line 20, the instruction PRINT" causes three lines to be skipped – note that this is three apostrophes, not a quote and apostrophe ("') or an apostrophe and a quote ('). Finally, line 30 shows what happens when the apostrophe signs are used within the quote mark. The items which are enclosed between quotes are printed as shown, but all the marks like apostrophes, colons, semicolons, and commas have quite different effects when they are *not* enclosed between quotes. As we shall see, letters also have quite different effects when they are not enclosed by quotes.

```

10 PRINT "Line 0"; "";"Line 3"
20 PRINT
30 PRINT "This is 'apostrophe"
"

```

| |
|----------------------|
| Line 0 |
| Line 3 |
| This is 'apostrophe' |

Fig. 2.8. Using the apostrophe to print a blank line.

Multi-statement lines

If Spectrum is your first computer, the heading may not be of much interest, but if you have graduated from the ZX81, then it should. Multi-statement lines mean that one line number can be followed by several separate instructions. These still need to be separated, but we can use a colon (:) as a separator rather than starting a new line. Using the colon in this way lets us group a set of instructions together so that they are easier to follow, and it also saves memory, because each time you start a new line you need to use several units of memory (called *bytes*). Each new line uses up to 5 bytes of memory, but if we put several 'statements' (complete instructions) on each line, the only 'waste' is the colon, which uses just one byte of memory. Even with the size of memory you have in the Spectrum, you may need to economise on memory now and again.

```

10 PRINT TAB 12;"Example": PRINT
"all of this has been program
med": PRINT "into a single line"

```

Fig. 2.9. A multi-statement line – each part is separated from its neighbours by colons.

Figure 2.9 shows an example of a multi-statement line. The 'statements' are all PRINT instructions and text. There are three sets of these statements in line 10 – and line 10 is all of the program! It looks unwieldy, but it can be useful. In later chapters, you will see this used to place all the instructions for a program into one single line.

Chapter Three

Introducing Variables

We saw in Chapter 2 that we could carry out simple tasks of printing messages and the results of arithmetic by using the PRINT instruction. As we used it, however, it was rather elementary, and it needed a lot of typing. Take a look at Fig. 3.1 – it's a

```
10 PRINT "Donald Smith"
20 PRINT "Kirstie Smith"
30 PRINT "Gordon Smith"
40 PRINT "Sarah Smith"
```

Fig. 3.1. A program that prints names – the hard way!

program that prints four names. As it stands, it requires the word SMITH to be typed four times. This risks wearing out your fingers and making your thumb numb, and the computer can tackle work like this in a much better way. Figure 3.2 shows how – we use a shorter 'code name' in place of SMITH, and wherever the code name occurs, the computer will put in SMITH for us. The result is that we

```
10 LET s$="SMITH"
20 PRINT "Donald ";s$
30 PRINT "Kirstie ";s$
40 PRINT "Gordon ";s$
50 PRINT "Sarah ";s$
```

Fig. 3.2. The easy way for name printing – a variable has been used to avoid repetition.

type SMITH only once – quite a saving if you are compiling a list of names.

The code that we have used is s\$, pronounced 'es-string', and the dollar (or string) sign is used to indicate that the quantity is not to be treated like a number. The letter s is called a *variable name*, because we don't have to use it only for SMITH, we can use it for ROBERTSON, BROWN, JONES or anything else we like. This type of code name (or variable) is a string variable name, and it must consist of a single letter and the \$ sign. The letter can be lower or upper case – the computer will take A\$ as being the same as a\$.

Looking now at line 10 in Fig. 3.2, the word LET performs what we call an *assiguation*. LET s\$ = "SMITH" means 'make s\$ represent the word SMITH', and for as long as this program is in the computer, s\$ will do just that, so that the command PRINT s\$, in or out of a program, will cause SMITH (no quote marks, notice) to come up on the screen. We can alter this by clearing the program out, using the NEW key followed by ENTER, by switching off and on again, or by changing the assiguation of s\$ to something else. Try typing this extra line:

```
35 LET s$ = "PLINGE"
```

When you have entered this, LIST, and you will find that the line has been slotted into its correct place between 30 and 40 (now you know why we number lines in tens). RUN this program, and you'll see what I mean when I say that s\$ has been re-assigned in line 35.

We can assign a variable name to a number as well, using statements like LET a = 14 or LET quantity = 56. There's no dollar (string) sign added when the number is *just* a number, but if we wanted to code the phrase 14 Acacia Avenue, then we would need to use a string variable, LET a\$ = "14 Acacia Avenue", because there is more than just a number here. We could, if we liked, use a string variable to code a number – LET b\$ = "14" – so why do we use different methods?

The answer is that numbers might be used in arithmetic. You can multiply 3 by 6, but you don't multiply "14 Acacia Avenue" by "3 Pints, please". We use simple number variables for numbers on which we are going to use some form of arithmetic or even, perish the thought, mathematics. We use string variables for everything else. Number variable names, unlike string names, can consist of more than one letter, and can also be in upper or lower case.

The reason for the difference is the way that the values assigned to these variables are stored in the memory. When a string is stored, each character in the string of letters or numbers is stored as a number code, using what is called ASCII code (American Standard for the Coding and Interchange of Information). Figure 3.3 shows what these ASCII codes are for Spectrum – they start at 32, which is a space, and end with 127, which is the copyright sign. This is an international code, but most computers use some number codes differently, and no other computer, for example, uses the copyright sign. Each character in a string means one code number, one byte of memory used up. A number can also be stored in this way, but there is a much more economical method available for storing numbers, which has the advantage of making arithmetic easy. That's why we use both methods, and later on we'll look at methods of converting one type of variable into another.

Input

So far, everything that we've typed into a program has been put there from the start, as part of the program. This cramps the computer's style, and to give you more choice, there is a way of putting information into your program while it is running. This is done by assigning a value to a variable name from the keyboard rather than by using LET. An example should make it clearer – type the lines of Fig. 3.4 and RUN the program. line 10 prints 'What surname?' on the top part of the screen. The words that you type, your input, appear on the bottom line, with the familiar flashing L which appears when the computer is awaiting letters. You can now type a name – but you don't need any quote marks. The computer has been warned by the instruction word INPUT that what you type will be assigned to a variable, and since the variable is a string variable, s\$ in this example, the quotes are supplied by the computer. The word that you type is not actually assigned to s\$ until you press the ENTER key, so you have time for second thoughts (or third, or fourth). When you press the ENTER key, the assignment is

| | |
|-----|----|
| 32 | |
| 33 | ! |
| 34 | !" |
| 35 | !" |
| 36 | !" |
| 37 | !" |
| 38 | !" |
| 39 | !" |
| 40 | !" |
| 41 | !" |
| 42 | !" |
| 43 | !" |
| 44 | !" |
| 45 | !" |
| 46 | !" |
| 47 | !" |
| 48 | !" |
| 49 | !" |
| 50 | !" |
| 51 | !" |
| 52 | !" |
| 53 | !" |
| 54 | !" |
| 55 | !" |
| 56 | !" |
| 57 | !" |
| 58 | !" |
| 59 | !" |
| 60 | !" |
| 61 | !" |
| 62 | !" |
| 63 | !" |
| 64 | !" |
| 65 | !" |
| 66 | !" |
| 67 | !" |
| 68 | !" |
| 69 | !" |
| 70 | !" |
| 71 | !" |
| 72 | !" |
| 73 | !" |
| 74 | !" |
| 75 | !" |
| 76 | !" |
| 77 | !" |
| 78 | !" |
| 79 | !" |
| 80 | !" |
| 81 | !" |
| 82 | !" |
| 83 | !" |
| 84 | !" |
| 85 | !" |
| 86 | !" |
| 87 | !" |
| 88 | !" |
| 89 | !" |
| 90 | !" |
| 91 | !" |
| 92 | !" |
| 93 | !" |
| 94 | !" |
| 95 | !" |
| 96 | !" |
| 97 | !" |
| 98 | !" |
| 99 | !" |
| 100 | !" |
| 101 | !" |
| 102 | !" |
| 103 | !" |
| 104 | !" |
| 105 | !" |
| 106 | !" |
| 107 | !" |
| 108 | !" |
| 109 | !" |
| 110 | !" |
| 111 | !" |
| 112 | !" |
| 113 | !" |
| 114 | !" |
| 115 | !" |
| 116 | !" |
| 117 | !" |
| 118 | !" |
| 119 | !" |
| 120 | !" |
| 121 | !" |
| 122 | !" |
| 123 | !" |
| 124 | !" |
| 125 | !" |
| 126 | !" |
| 127 | © |

Fig. 3.3. A list of ASCII codes as used in the Spectrum.

```
10 PRINT "What surname?"
20 INPUT s$: CLS
30 PRINT "Tom ";s$;" Jim ";s$;""
40 PRINT "June ";s$;" Laura ";s$
```

Fig. 3.4. Using INPUT, which allows you to type something into the computer while a program is running.

carried out, so that the name you have entered is coded as s\$, and used in line 30 – note the use of apostrophe marks to guide the printing, and the use of CLS(Clear Screen) to make sure that the screen is not cluttered with the words 'What surname?' when you print the names.

Operating on number variables

Before we start to look at some of the remarkable things that we can do with string variables, we need to spend some time with number variables. Very few programs exist that do not make some use of number variables, and since arithmetic is one of the important uses of these variables, we'll start with some arithmetic.

We've already met the five arithmetic signs, $+$ $-$ $*$ $/$ and \downarrow in little pieces of arithmetic in which only one of these actions was used at a time. An arithmetic expression is a set of operations such as $(3-2)*6/4$, and it's important to know what the computer does with expressions like this. If you make any use of a calculator, you may already know that entering an expression like $3-2*6/4$ gives a result which is not the same as the result of $(3-2)*6/4$. The reason is that machines have to follow set rules, and the rules that most calculators and practically all computers follow is called the rule of precedence. Operations, like addition, division, multiplication and subtraction, are arranged in order of precedence, meaning that some will be done before others. The order is one which has been used for centuries, long before calculators were invented. It is MDAS – multiplication and division are carried out before addition and subtraction. The other one, raising to a power (sometimes called exponentiation) comes before multiplication.

An expression like: $20 - 6*2 + 4$ therefore gives 12 as its result. Multiplication is done first, so $6*2 = 12$. Addition and subtraction are then carried out: $20 - 12 + 4 = 12$. We can change the order by putting brackets round any part we particularly want to be done first, so that if we wrote $(20-6)*2+4$, the result would be 32. Why? Well, $20 - 6$ is 14, $14*2 = 28$, and $28 + 4 = 32$.

What makes the computer so useful, though, is that you don't have to put actual numbers into expressions at the time when you write the program. You can write the program using variable names for the numbers, and then assign the variable names to actual numbers using INPUT. Try the simple example in Fig. 3.5. A

```

10 PRINT "A number, please...""
20 INPUT n: CLS
30 PRINT "The square of ";
" is ";
40 PRINT "The square root of ";
" is -" TAB 10: SQR n

```

Fig. 3.5. Using a number variable in a square-and-square root program. Don't use negative numbers – they have no real square root, and Spectrum refuses to calculate the square root of a negative number!

number is asked for in line 10, and the INPUT in line 20 then assigns the variable name n to whatever number you type when the program runs. Line 30 then prints the square of this number, and line 40 prints its square root, using SQR. To get the SQR instruction, press both shifts and release, then press key H. SQR finds the square root of a number, whether it is a number assigned to a variable name or just a number, assuming that there is a square root. Negative numbers do not have real square roots, so if the number that you enter in line 20 is negative you will get an

error message which reads 'A Invalid Argument'. What this means is that the number assigned to the variable is one that can't be used by the SQR instruction, so that the computer can't carry on. In Chapter 4 we'll see how you could alter this program so as to avoid the error message, but for the moment, we'll continue to explore the INPUT instruction. Note, incidentally, in this program, that the words between quotes are printed, but when a word is used as a variable name for a number, then the *number* is printed.

As we've used it in line 20 of Fig. 3.5, the INPUT instruction has caused the computer to stop and wait for you to type a number and press ENTER. You can enter more than one item into a program by an INPUT instruction. Take a look at Fig. 3.6. The request printed in line 10 is for a distance in metres and centimetres. When you reply (line 20) you must type the number of metres, press ENTER, and then type the number of centimetres (which will appear half-way along the bottom line) and then press ENTER again. Line 30 then converts the length to centimetres by multiplying the number of metres by 100 and adding the number of centimetres, and then converts to feet by dividing by 30.48.

```

10 PRINT "Type length in metres
 and centimetres"
20 INPUT m, cm: CLS
30 LET f = (m*100+cm)/30.48
40 PRINT "that's "; f; " feet"

```

Fig. 3.6. A conversion program – metres and cm to feet.

We don't have to stick to number variables when we INPUT, either. Figure 3.7 shows an example of a program which allows you to enter your name, age and year. Your name has to be assigned to a string variable – it can't be stored like a number, though age and year can. If you get it wrong and type your age first when the program runs, the computer will assign your age to the variable name n\$, but it

```

.. 10 PRINT "Name and age, please-
.. 20 INPUT n$, age
30 PRINT "... and what year is
this?"
40 INPUT year: CLS
50 PRINT "Well, "; n$; ", do you
realise"; "you will be "; 2000-year
" in the year 2000?"
```

Fig. 3.7. Using string and number variables in the same INPUT.

will not accept your name as an answer for the variable called 'age', because a name is not a number, though a number can be stored as a string. If you try to enter a string (a name or any combination of letters with digits) when a number is requested, then the computer will refuse to accept the input by printing an error message like:

2 Variable not found 20:1

meaning that this is a type 2 fault (wrong type of variable) in line 20, part 1. The computer tries to help you by showing quotes when it expects a name to be input, but not when a number is expected.

Now look at another twist to the INPUT instruction. You can use INPUT to do the action of PRINT along with its own action, putting a message to be printed between quotes just as you would in a PRINT instruction. Figure 3.8 shows an

```

10 PRINT TAB 9;"MULTIPLICATION
" 20 INPUT "...A number,please-",,
" 30 INPUT "...and another,please
" 40 CLS : PRINT "The product is
";a*b

```

Fig. 3.8. Typing messages as part of the INPUT instruction.

example in which you are asked to enter two numbers. The messages in lines 20 and 30 are part of the INPUT instructions, separated from the variable names (a and b) by the quotes and the comma. You can have several of these messages and variable names in one INPUT line, as Fig. 3.9 shows. There is another variation of

```

10 INPUT "Please type the day"
;d$;"date";date;" and month"
;M$ 20 PRINT "d$;" "date;" " ";M$
```

Fig. 3.9. Mixing messages and variable names.

INPUT which uses INPUT LINE, and which will not show quote marks for a string input, but you don't need to worry about this one at the moment. An example is shown in Fig. 3.10.

```

10 PRINT "Please type item, the
" page" 20 INPUT LINE i$: CLS
30 PRINT "The index item is-"
40 PRINT TAB 12;i$
```

Fig. 3.10. Using the INPUT LINE instruction.

String slicing

It sounds like something you do with beans, but it's a lot more useful. String slicing means selecting part of a string to use for some other purpose, and the selection is done by counting the letters of the string, starting with the first one as you might expect.

Take a look at the example in Fig. 3.11. It's simple enough – it asks for your name, and then presents you with some letters selected from the name. The important thing to understand is how these letters have been selected. A single

```

10 INPUT "Your name,please";N$
20 PRINT ""The first letter"
s-=";N$(1)
30 PRINT ""The second letter
is-";N$(2)
40 PRINT ""The first three le
tters are-";N$(1 TO 3)

```

Fig. 3.11. Slicing a string.

letter is selected by using a single number inside the brackets, and the number has to be the position number of the letter in the string, 1 for the first, 2 for the second, and so on. To select more than one letter we need a start number and a finish number, with the instruction word TO (on the F key) placed between them. The instruction N\$(1 TO 3) means select letters 1, 2, and 3 from N\$; N\$(1 TO 5) would mean selecting letters 1 to 5 inclusive, and so on. If you miss out the first number (as in N\$(TO5)), the computer will take it that the first number is 1; if you miss out the second number, the computer will take it that you want to go to the end of the string.

```

10 INPUT "Please type a five-
letter word";N$
20 CLS : PRINT TAB 15;N$(5)
30 PRINT TAB 14;N$(2 TO 4)
40 PRINT TAB 13;N$
```

Fig. 3.12. String slicing as a decorative art!

Figure 3.12 shows another example of string slicing in action. At this stage, however, it's not so easy to show how useful these actions are, because you don't know enough about BASIC yet to follow what is going on in a program that makes a lot of use of string slicing, but you should be able to see how it can be used in a variety of tasks. Picking out words with the same initial letter, for example, is easy, as is finding a name in a list. Not-quite-so-easy tasks include finding if one word is included inside another, or how many letter 'es' are used in all the words of a text, and there are countless other examples, some of which we shall use as we continue in this book. It's also possible to put new letters into a word by using the slicing commands!

Other string actions

Slicing is just one of the actions that can be carried out on a string variable, and which can't be carried out on a number variable. LEN is one of the instruction words for these actions which are often called 'string functions'. A string function is an instruction word that needs a string or string variable name (called the 'argument') to work on. Take a look at Fig. 3.13, which shows how LEN can be used. It's particularly useful when you want to slice a string but you don't know the length of the string, and we'll be looking at that type of situation later on.

Having sorted out LEN, we can look now at some other string functions. VAL

```

10 INPUT "Please type a word";
20 LET n$="String function"
30 CLS : PRINT n$;" has ";LEN
n$;" letters"
40 PRINT ". and ";n$;" has ";LEN
n$;" characters"
50 PRINT "Remember that each character is ";
"counted, including spaces"

```

Fig. 3.13. Using LEN to find the length of a string.

is a particularly useful one, because it can extract a number from a string. On some computers, VAL will extract a number from a string that contains a number preceding some other character, but it doesn't work that way on Spectrum. The string must consist of just a number or of some arithmetical expression which will result in a number, like $4 + 2.6$ or $2.7 * 1.4$. Figure 3.14 shows some of the things

```

10 LET a$="24"
20 LET b$="1.56"+"*"+"2.04"
30 PRINT a$,b$
40 PRINT VAL a$,VAL b$
50 PRINT VAL a$+14

```

Fig. 3.14. Joining strings and using VAL.

you can do. To start with, line 20 uses the + sign between sets of strings. When the result, which is b\$, is printed in line 30, you can see that the strings have been run together. This is a special string action called *concatenation*, that is carried out when the + sign is placed between two or more strings, and its action is nothing like the action of + for two numbers. Typing PRINT 24 + 32, for example, gives 56, but typing PRINT "24"+"32" gives the answer 2432. Once again, strings are not treated like numbers. The use of VAL, however, will convert a string which consists of digits or an expression, into a number, as lines 40 and 50 of Fig. 3.14 shows. You could also use a number variable to 'catch' the number that VAL extracts from the string by having a line like:

```
35 LET n = VAL a$
```

and you could then use n as a number variable – its value in our example would be 24. What you can NEVER do, however, is to mix strings and numbers.

Because of these differences – we can perform arithmetic with number variables and operations like slicing with string variables – VAL is useful for converting a string which consists of digits back into number form, and there's an opposite action which is carried out by using the instruction word STR\$. STR\$ will convert a number into string form, so that you can slice it (very useful for removing unwanted numbers after the decimal point, for example), and do all the things that you can do with strings. Figure 3.15 shows an example which can be useful – it rounds off a number by converting it to a string and slicing it. Line 40 then shows another way, however, which does not make use of string slicing. It multiplies the number by 100, takes the whole number part of the result, and then divides by 100

```

10 LET c$=STR$ 2.7614
20 CLS : PRINT c$( TO 4)
30 LET n=2.7614
40 PRINT INT (n*100)/100

```

Fig. 3.15. Rounding off a number in two different ways.

again. Remember when you disentangle this one that whatever is innermost in brackets is done first. The instruction INT means *take the integer part*, cutting off the fractional part following the point. This is as near as we get to a slicing action for numbers.

CODE and CHR\$

CODE and CHR\$ are another two related instructions which have opposite actions. Each character in a string is stored in the form of a number code, its ASCII code number. CODE is used to find what the ASCII code is for the first letter in a string. CHR\$ does the opposite – it gives the letter equivalent of a number if there is such a number in the ASCII code. The example of Fig. 3.16

```

10 LET a$="Sinclair"
20 PRINT CODE a$
30 PRINT CHR$ 68
40 PRINT CHR$ 138

```

Fig. 3.16. Using CODE and CHR\$.

shows both CODE and CHR\$ in action. In line 20, what is printed is the ASCII code for S, since this is the first letter of Sinclair. If a\$ had been blank, the number printed in line 20 would be zero – we talk about a function *returning* with something so that we could say that the CODE function would return with a zero for a blank string. The instruction word CODE is also used in a different way along with LOAD and SAVE, but we're not concerned with that at the moment. In line 30, the character corresponding to ASCII 68 is printed (it's D), and in line 40, the character corresponding to code number 138 is printed. ASCII character codes go up as far as 127 only, and anything beyond this is used in different ways by different computers. As it happens, 138 is one of the codes for the graphics characters of Spectrum. CODE, incidentally, as we have used it here, is the equivalent of the ASCII instruction which is used in other computers.

Comparing strings

We saw earlier that we can join (concatenate) strings by using the + sign. Some of the other mathematical signs can also be used to compare strings, in particular the equality and inequality signs. As far as strings are concerned, the equality sign means 'identical to', not just equivalent to. You can write the number 100 as 100, for example or as 1E2 (1×10^2), and the computer correctly accepts these as being

the same. It will not accept that WORD is the same as word or Word, however, because of the difference between the ASCII codes for upper case and for lower case letters. W has the ASCII code of 87, w has the ASCII code of 119, so the two are not identical. When the computer compares strings, it compares character by character, using the ASCII codes. If the first characters are equal, then the second characters are compared and so on until a difference is found or complete identity checked. The mathematical signs $>$ (greater than) and $<$ (less than) apply also to the ASCII codes. The letter w is taken as being *greater than* the letter W, because its ASCII code is greater; the letter A is *less than* M because its ASCII code is smaller. When complete words are compared, the computer will check character by character just as it does when checking for equality.

These signs can be combined, so that \leq means less than or equal to, and \geq means greater than or equal to, and \neq means not equal to. Each combination of these signs is found on a separate key of Spectrum, so that you don't have to type them separately.

How do we use these equalities and inequalities? Next chapter, please!

Chapter Four

Loops, Repetitions and Arrays

So far, we have looked at the type of program that is called linear. A linear program has a start, does something once, and then stops. A linear program is simple and easy to follow, but the main advantages of using a computer start to appear when we make programs follow different paths for different cases and repeat steps. In this chapter, and from now on, *you will see the printed programs using upper case only for instruction words (which you don't need to type) – the words and letters that you need to type will be in lower case, or in a mixture of upper and lower case*. We shall also look at how programs are ended – a simple linear program ends when there are no more lines to carry out, but this may not be true of the types of program that occur in this chapter.

A program that can carry out different steps for different cases is called a branching program, and Fig. 4.1 shows an example. You are asked a question to

```

10 CLS : PRINT "IS YOUR NAME S
mith?"
20 INPUT "(Answer y or n, press
ENTER)":a$:CLS
30 IF a$="y" THEN GO TO 50
40 PRINT AT 10,1;"Well, we can'
t all be Smiths."; GO TO 9999
50 PRINT AT 10,1;"That's funny
,I know somebody","of that name."

```

Fig. 4.1. A branching program – line 30 decides what line will be performed next.

which you are supposed to give a reply y (yes) or n (no) by typing the letter, followed by pressing the ENTER key. Your reply is allocated to the variable a\$, and in line 30, a\$ is tested to see if it is "y". If it is, the last part of line 30 will cause the computer to jump to line 50, ignoring line 40. This is one of the few examples in which a computer carries out instructions which are *not* in strict line order. If a\$ is "n", or anything which is NOT "y", then line 40 is carried out, and since line 40 ends with GOTO 9999, line 50 can't possibly be carried out. When you type GOTO followed by the number of a line that does not exist, the computer will stop at that point – the reason for using 9999 is that it is the highest line number that Spectrum can use, so it's not likely to be an actual line number. Some computers have the instruction word END; Spectrum does not.

By using the IF test, then, the computer can distinguish between two answers and act on them in different ways. Notice how the test is formed – it starts with IF, and this is followed by the test of equality or inequality, and then by what has to be

done when the test succeeds THEN GOTO 50. Notice, incidentally, that line 50 is used only for a "y" answer, line 40 will be carried out for any other answer.

Figure 4.2 takes a different example. You are asked to type a number, which is then allocated to the variable name n. In line 20, the test is made using $n/2 =$

```

10 INPUT "Type a number, please
..";n
20 IF n/2=INT (n/2) THEN PRINT
  n;" is an even number.": GO TO
10
30 PRINT n;" is an odd number.
": GO TO 10

```

Fig. 4.2. Odd or even program – line 20 decides the difference.

$\text{INT}(n/2)$, which succeeds (*is true*) only if n is an even number – note that the brackets are essential. If n is odd, like 5, then $n/2$ contains a fraction (2.5 if n is 5), and INT will remove this fraction. In our example, this would give $2.5 = 2$, which is not true, so that the test fails for odd numbers. There's a sting in the tail (or in two tails), because after each printout, odd or even, the instruction GOTO 10 sends the program back to line 10 to request another number. This is an example of a loop, a piece of program that repeats, and in this case the loop is endless; as long as you like to keep typing numbers, the computer will test them and report to you whether they are odd or even. To break this particular loop, you have to type a letter (NOT a digit) and ENTER it. Normally, loops can be broken by pressing CAPS SHIFT along with SPACE (BREAK), but this method does not work when you try to break at an INPUT.

The advantage of using a loop is that it saves an immense number of program lines, since the same lines can be used over and over again. The disadvantage is that loops can be endless, so that when we put a loop into a program we must think in advance how we are going to stop it. We can either design it to loop for a specified number of times, or to stop when some specified input is received, and both of these methods are used extensively in BASIC programs.

Taking the second type first, look at the example in Fig. 4.3, which contains a

```

10 CLS : PRINT TAB 8;"Cumulati
ve Total": LET n=1: LET total=
0
20 PRINT "Enter a number when
instructed": "and press ENTER. Th
e computer": "will add the number
s and print a": "total until you
enter a 0."
30 INPUT ("No.":n;" ")
40 LET total=total+item: LET n
=n+1: PRINT "total so far is "; t
otal
50 IF item<>0 THEN PRINT "next
": GO TO 30
60 PRINT "end of entries."

```

Fig. 4.3. Keeping a running total. Using a name like 'total' for a number variable reminds you of how this name is being used.

number of new items for your attention. The program accepts numbers, and keeps a running total, adding each number to the previous total. It stops only when a 0 is entered, and unlike most of our previous programs, it requires some planning. Since an input from the keyboard needs to be prompted with some reminder (the flashing cursor isn't enough), the number of the item is printed, and since we want to start with Item 1, a variable n is set to the value of 1 in line 10. We also want to start with a total of zero, so the variable called 'total' (a number variable can be of a number of letters, remember) is set to zero, also in line 10. These actions in line 10 are called 'initialising variables', and they are very important when we start to deal with longer and more complicated programs, because we must be sure what values variables have when the program starts.

Line 20 contains instructions. Every program needs some instructions – even if you wrote the program you may have forgotten how to use it when you come to run it again. Note that all of the instructions are in one program line with the apostrophe used to force the printing to move to the next line. This is a special feature of Spectrum, and it makes it possible to design very neat printed lines with very little effort – just make sure that the end of a line does not extend further along the screen than the start of the last line (at a quote mark).

Line 30 asks for an input, and it's here that we encounter a new variation on INPUT that is unique to Spectrum. By putting "No.";n within brackets, we force the program to print out *No.* followed by the *value that n has at that instant*, which will be 1 on the first time round. If we omitted the brackets, the computer would take *n* in place of *item* as the name of the variable which was to be used to store the number. By using the brackets, the value of n is printed, and the variable name that is used for the number you enter is *item*, as intended. Note the space after n, so that the number which you type is not placed right up against the value of n.

In line 40, a new total is made from the previous total plus the new item. On the first time round, the variable *total* started with the value 0, so that the new value of *total* is now the same as the value of the variable *item*, the number that you typed. The use of the equality sign here is a bit misleading. It really means 'becomes' when used in this way – the new *total* becomes the value of the old *total* plus the value of *item*, and this is where the cumulative part comes in. There is a similar use of = in the next part of line 40, where we have $n = n + 1$. In this case, it simply means that the number allocated to n is increased by 1 – the new value of n equals the previous value + 1. The last part of line 40 prints the value of the variable *total* so far, and we're ready for the test in line 50. If the item which was entered is NOT zero (tested by item $\neq 0$), then the program loops back to line 30, with the word *next* being printed as a reminder. Each time the loop is used (on each *pass* through the loop, as we say), the value of the counter n is increased by 1, and the value which has been entered for the variable *item* is added in to the variable *total* and printed. When a 0 is entered, the program stops at line 60. Note, incidentally, that if we wanted to use some number other than 0 (like 999) to stop the program, we would have to test for the number *before* it was added to the total. You might like to consider how the program could be changed to do this.

Feeding a loop

When a program contains a loop, it has to be because there is some reason for repeating a set of lines. One very good reason is the use of a pair of instructions which are found on many computers but which will be new to former ZX80 and ZX81 owners. They are READ and DATA. The READ instruction causes the computer to take an item from a list. These list items are indicated by the word DATA, so that the computer ignores any line which starts with DATA until it comes to the READ instruction. The items in the DATA line, which are separated by commas, are taken one by one in sequence, and if there is more than one line of DATA, the lines are also taken in order. If you attempt to read more items than you actually have as DATA, then the computer will stop with the message 'Out of Data' displayed.

Figure 4.4 shows this useful set of instructions at work, and it also introduces

```

10 CLS : PRINT TAB 10;"Month-f
inder": : LET j=0
20 INPUT "Please type the numbe
r of the month.. ";n: LET n=INT
n
30 IF n>12 OR n<1 THEN PRINT "
Silly - try again": GO TO 20
40 LET j=j+1: READ m$: IF j>12
THEN GO TO 40
50 PRINT "Month number ";n;" is "
;"m$"
60 DATA "January","February",
"March","April","May","June","Jul
y","August","September","October",
"November","December"

```

Fig. 4.4. Using READ...DATA in a month/number program.

some points of programming that we haven't met so far. The aim of the program is to print the name of a month when its number (1 to 12) is typed in at line 20. What you have to watch in any program of this type is what can happen when any user (which might not be you!) types a ridiculous number, like 2.6, or -16, or 23. At the end of line 20, the problem of fractions is dealt with by the statement LET n = INT n, which makes the new value of n equal to the whole number part of its previous value, so that 2.6 would become 2. Line 30 then sorts out other silly inputs, numbers greater than 12 or less than 1 (negative). If such numbers are used, a message will be printed, and the program returns to line 20 for another attempt. If the number is reasonable, then line 40 is the next one used. The first step, LET j = j + 1, bumps up the value of the counter j by 1 (from 0 to 1 first time round), and the READ instruction makes m\$ equal to a month name – January on the first time round. Note that you need quotes around the words in the data list because they are going to be assigned to a string variable – some computers do *not* need quotes. Spectrum needs quotes for the items that are to be assigned to string variables, but not for items that will be assigned to number variables.

The third part of line 40 then compares the value of i with the value of n , and if

they are equal, the rest of line 40 will be ignored and in line 50 the message is printed with the name of the month. If the value of n was not equal to the value of j, however, the computer makes line 40 repeat, bumping up the value of j and comparing again. This continues until a match is found. By ensuring that the number which is entered lies between 1 and 12, we can be certain that the numbers will eventually match, so we don't need any further tests to end the loop.

These tests of the number entered in line 20 are essential to the program, and are called 'mugtraps' – from now on we shall have to pay more attention to such traps, which help to make programs easier to use, though more difficult to design. Incidentally, be very careful when you put in the word READ as you are programming – it's obtained using the A key, which also carries the NEW instruction. If you forget to press the shifts before hitting the A key, you may get NEW, whose effect is to wipe the program from memory!

The loops we have looked at so far depend on the use of GOTO, which is one of the least satisfactory instructions in BASIC. Like whisky, GOTO is useful at times, but causes problems when used in excess. A program written with a lot of GOTOS, especially when they refer to lines that are spaced far away in the program, is difficult to understand, difficult to work on, and can often suffer from strange faults which are difficult to trace. Because of this, other ways of forming loops and ending them are important, and the one that is used on Spectrum is the FOR ... TO ... NEXT loop.

FOR ... TO ... NEXT is a set of instructions which will cause a loop to be carried out for a fixed number of times. This number can be placed following the TO, or a number variable can be used. In addition, another number variable has to be used to count the number of times that the loop operations are carried out, and this variable name is placed between the FOR and the TO parts of the instruction. For example, if you want to start counting at 1 and end with 6, you might use FOR n = 1 TO 6; if you want to start with 5 and end with 10, you might use FOR n = 5 TO 10. You could also use instructions like FOR n = 1 TO j or FOR n = 1 TO LEN a\$.

Traditionally, we use letters j, k, and n for counter variable names, but you can use whatever you like provided that you are not also using the same variable name for something else. The count for the FOR ... TO ... NEXT loop will be in steps of 1 unless you add another word, STEP with a number following it. You can, for example, use FOR n = 1 TO 9 STEP 2 to count 1, 3, 5, 7, 9, or FOR n = 8 TO 1 STEP -1 to count 8, 7, 6 ... down to 1. At the end of the loop, after all the instructions that are to be carried out in each pass through the loop, you have to put NEXT n (or whatever variable name you used) to instruct the computer that this is where the loop ends and returns for the next pass.

Now let's take a look at a program in Fig. 4.5 that uses a FOR ... TO ... NEXT loop, along with a new use of READ ... DATA and a few other tricks. It's by far the longest program that we have used for illustration so far, and it merits careful study, because it illustrates a lot of useful 'how-to-do-it' tips. The aim is to print the name of a day, given the input of a day number, 1 to 7. As usual, lines 10 to 30 deal with the input and test the number which is allocated to the variable name *day* for

```

10 CLS : PRINT TAB 10;"Day of
week";
20 PRINT "Please type day num-
ber (1 to 7)"
30 INPUT day: LET day=INT day:
IF day<1 OR day>7 THEN PRINT "S-
illy value, try again.": GO TO 20
40 PRINT "Would you like-"; TAB 5;"1. En-
glish."; TAB 5;"2. Fren-
ch."; TAB 5;"3. Spanish."
50 PRINT "Type digit-don't use
ENTER": PAUSE 50
60 LET k$=INKEY$: IF k$="" THE
N GO TO 60
70 LET d=VAL k$: IF d<1 OR d>3
THEN PRINT "Sorry, 1 to 3 only!
try again": GO TO 60
80 RESTORE 110#d
90 FOR j=1 TO 7: READ w$: IF j
<>day THEN NEXT j
100 PRINT "Day ";day;" is ";w$;
110 DATA "Monday", "Tuesday", "We-
nesday", "Thursday", "Friday", "Sa-
turday", "Sunday"
220 DATA "Lundi", "Mardi", "Mer-
ci", "Jeudi", "Vendredi", "Samedi",
"Dimanche"
330 DATA "Lunes", "Martes", "Miér-
coles", "Jueves", "Viernes", "Sabad-
o", "Domingo"

```

Fig. 4.5. READ...DATA extended by using RESTORE. The language is selected by RESTOREing to different data lines.

silly values. Line 40 gives a new twist to our list, a choice of day name in three languages. This is done by choosing the name from three separate lists, and the neat method of choosing the list is something that is a feature of Spectrum. You are asked to select the language by choosing a number between 1 and 3, and advised NOT to use ENTER. The reason is that we've used a new instruction INKEY\$ in place of boring old INPUT. INKEY\$ is a peculiar instruction – it is a test for a key which happens to be pressed at the time when the instruction INKEY\$ is carried out. Now you would have to be very smart with your fingers to catch it at the right instant, so INKEY\$ is always used in a loop, such as we have in line 60. We assign another string variable, k\$, equal to INKEY\$, and if the string is blank (equal to ""), as it would be if no key happens to be pressed, then the line keeps repeating until a key is pressed. A minor detail here is that if we have just pressed ENTER before the INKEY\$ line, the test may detect the ENTER key. In this program, we get around this by PAUSE 50, which makes the computer wait for one second before carrying out the INKEY\$ instruction. Later on, we'll look at other solutions to this problem.

The value of the variable k\$ then has to be tested – you might have pressed a letter key or the wrong number – and this mugtrapping is done in line 70. Notice that we have used VAL to convert the value of k\$ into number form to carry out the tests. Moving on, line 80 contains another new instruction, RESTORE. It means restore the list of DATA to the start, but Spectrum, unlike older designs of

computers, allows you to restore to the start of *any* data line that you like to choose by having a number *or* *number variable* following RESTORE. By using RESTORE 110*n, we restore to line 110 if n = 1, to line 220 if n = 2, and to line 330 if n = 3; this is how we select the correct language list. From now on it's plain sailing over more familiar waters as we run through the list, counting the days until we end up with the selected one. We can use the FOR j = 1 TO 7 loop here because we know that there are only seven days in the week, even in other languages!

Wherever we know how many times a loop needs to be performed (the number of *passes* through the loop), or we can instruct the computer to find this number, then we can make use of the FOR ... TO ... NEXT loop. Instructing the computer to find the number may sound unusual – it isn't, and take a look at Fig. 4.6 to see what I mean. By this stage you should be able to follow what's going on in lines 10 to 30. The name of the day is typed in English – perhaps we should have reminded the user that the day name should start with a capital letter? The FOR ... TO ...

```

10 CLS : PRINT TAB 10;"TRANSLA-
TION"
20 PRINT "Please type name of
day in full.": RESTORE 100
30 INPUT d$: FOR n=1 TO 7: RE-
D e$: IF d$<>e$ THEN NEXT n: PRI-
NT "I have no record of ";d$;""
"Please try again.": RESTORE 10
@: GO TO 30
40 RESTORE 200: FOR n=1 TO n:
READ g$: NEXT n
50 PRINT d$;" in German is ";g
$;
100 DATA "Monday", "Tuesday", "We-
nesday", "Thursday", "Friday", "Sa-
turday", "Sunday"
200 DATA "Montag", "Dienstag", "H-
ittwoch", "Donnerstag", "Freitag",
"Samstag", "Sonntag"

```

Fig. 4.6. A translation program that uses a FOR...NEXT loop to find a number.

NEXT loop in line 30 then runs through the list of English names looking for one that matches what you have typed. If no match can be found – and remember that Monday does not match MONDAY or monday – then the computer prints a message and asks you to try again. If a match *is* found, the counting of the loop is interrupted at some value of variable n which represents the number of places along the list of days at which the correct day was found. In line 40, the RESTORE 200 then shifts to the start of the German list, and the new loop uses n as its limit, so that n items will be read. Each of them is assigned to the string variable name g\$, but each is replaced by the next until all n items have been read and the loop stops. The word which is now stored as g\$ will be the word as many places along the German list as the day which you typed was along the English list, and it's printed in line 50.

One point here: This and the previous example contain FOR ... TO ... NEXT loops which have not been allowed to finish. Some computers do not allow this,

but Spectrum does, and it makes this type of program very much easier to carry out.

Arrays

We've seen how we can READ items into the computer with the READ ... DATA pair of instructions. So far, though, we haven't been able to use more than one item at a time. When we use a loop which contains a line such as:

50 READ m\$

then each time the loop is performed, a new value for m\$ is assigned. This is not always what we want, and we often need to be able to get hold of any one item from a list without having to read through the whole list. The trouble here is that each item needs a different variable name, and if we have to assign a different variable name for each we can't use a loop; we have to program it as:

READ a\$: READ b\$: READ c\$

and so on.

An array is a way round this difficulty. An array is a *numbered list* which uses the same variable letter (one letter only) for its name, but allocates a different reference number (called a *subscript*) to each one so as to separate the items. What makes it so useful is that the computer can use a number variable as the subscript number. If we have items a\$(1), a\$(2), a\$(3) ... (read as a-string-of-1, a-string-of-2, a-string-of-3 ...), then we can use a\$(n) to represent any of them, whichever one corresponds to the value of variable n. When we assign a value to n, we can select an item, so that if n = 1, then we can print (or otherwise use) a\$(1), and so on.

Before we look at an example, though, we have to consider some planning. Some computers will accept arrays with very little preparation, some with just a little more thought. The Spectrum needs a little more thought. You have to instruct the computer to lay aside some of its memory for the array *before* you start entering items into the array, and this is done in the program by using the instruction word DIM (no, it's not the opposite of bright!). DIM means DI^Mension, and for an array of strings it has to be followed by *two* numbers in brackets, separated by a comma. The first number is the number of items in the array, the second number is the maximum number of characters that will be used in any item. If your items are Horse, Cow, Pig, Hen, Dove, Dog, Cat, Goat, then you have 8 items, of which none has more than 5 characters (Horse is the longest) and you would type:

5 DIM a\$(8, 5)

to prepare storage space for an array called a\$(). If you don't do this, you will get the error message 'Variable not found' when you try to run your program.

Take a simple example in Fig. 4.7. We're using the list of animals that we mentioned above, so that the DIM instruction in line 10 is DIM a\$(8,5) – preparing for eight items of no more than 5 characters each. When the item

```

10 DIM a$(8,5): FOR n=1 TO 8:
READ a$(n): NEXT n: RESTORE
20 CLS : PRINT TAB 12;"ANIMALS"
" " "This Program Prints the name
" " "of" "a domesticated animal each
" " "time" "you press a key."
30 PRINT "It could be the star
" " "of a game."
40 LET n=1+INT (RND*8)
50 LET b$=INKEY$: IF b$="" THE
N GO TO 50
60 CLS : PRINT AT 5,12;a$(n):
GO TO 40
100 DATA "Horse","Cow","Pig","H
en","Dove","Dog","Cat","Goat"

```

Fig. 4.7. Using an array to store animal names.

contains fewer than 5 characters, the computer will *make its length up to five with blanks* so as to ensure that each item has the same length.

The rest of line 10 then reads an item from the list and assigns it to an array number – the loop starts with n = 1, so that item 1 is read from the DATA list and assigned to a\$(1), item 2 is read when n = 2, and so on. We don't do much with all this – yet. Line 40 picks a whole number value for n. It has to be a whole number and positive, because an array can't use items like a\$(1.5) or a\$(-6), and it introduces a new instruction, RND. RND causes a number to be generated by the computer, and the value of this number will be fairly unpredictable, like the number you get by throwing dice. It's almost a random number, hence the name RND for the instruction. By itself, RND produces a number which is a positive fraction, ranging from just over zero to just less than 1. If we multiply this number by eight, we will get values that range from around zero (because eight times almost-zero is still almost zero!) to just less than 8. INT then chops off the fractional part, giving numbers which range between 0 and 7. Add 1 to this, and we have a set of numbers ranging at random between 1 and 8, just what we need to select one of our animals. Line 50 then keeps the computer testing the keyboard, waiting for you to press a key, and when you do so, the animal name which has been selected by the randomly picked number is printed. We haven't done any more to the program, but it could just be the start of something!

A string array need not be filled by using READ ... DATA, of course, it could equally well be filled by using INPUT. A loop which starts:

FOR n = 1 TO 50

and contains:

INPUT a\$(n): NEXT n

will fill values into a string array, but you will have to dimension correctly and add your own mugtraps. If you have dimensioned your string by using DIM a\$(50, 10), for example, meaning that you will use 50 items of not more than 10 characters each, it's only too easy when INPUT is used to slip in a name which has 11 or more characters. When you use READ ... DATA you have to type in all the

names, and it's easy to see if one is going to be too long. When INPUT is used, you can only check with the help of LEN by using something like:

```
IF LEN a$(n) > 10 THEN PRINT "Name too long - please try again":  
GOTO
```

with the GOTO taking you back to the INPUT without altering the value of n. This way you can catch the error and sort it out before any problems are caused.

Number arrays

A number array is easier to arrange than a string array, because the computer takes the same amount of memory space to store any number. You will still have to use DIM, but with only one number, the number of items in the array. As usual, an illustration helps, and it also shows a detail of programming that we have not used before, a loop inside a loop, or a 'nested' loop as it is called.

Figure 4.8 is a program which prints multiplication tables (the things they used

```
10 PRINT TAB 15;"TABLES"  
20 PRINT "How many sets of tab  
les would you like to see?"  
30 INPUT n: LET n=INT n: IF n <  
1 OR n>10 THEN PRINT "Maximum of  
10, no silly answers!": GO TO 20  
40 DIM k(n): FOR j=1 TO n: PRI  
NT "Table-";  
50 PRINT "Please enter number  
whose table is wanted": INPUT k(  
j): NEXT j  
60 FOR j=1 TO n: CLS : FOR p=1  
TO 12  
70 PRINT k(j); " times ";p;" is  
";k(j)*p  
80 NEXT p: PRINT "Press any key  
for next table"  
90 LET k$=INKEY$: IF k$="" THE  
N GO TO 90  
100 CLS : NEXT j  
110 PRINT "End of TABLES"
```

Fig. 4.8. A name-and-number program with several new features.

to get kids to learn in the days when they wanted them to leave the school able to count). The program will print a set of up to ten tables for whatever numbers you like to enter. The number of tables has to be entered in line 30 so that the array can be correctly dimensioned and the FOR ... TO ... NEXT loop set up. If you enter 8, for example, the computer will prepare to print 8 sets of tables. There is a mugtrap for fractions and another one which refuses to accept negative numbers or numbers greater than 10. You can fix your own limits, obviously, by altering the number in line 30 from 10 to whatever you want.

Once this is fixed, the array is dimensioned. The numbers for which you want tables are going to be held in an array k(n), so this is dimensioned, and a loop FOR j = 1 TO n set up, because you want n sets of tables. The last part of line 60 is a new

piece of programming, though, a loop within the main loop. We're going to take each value of j and multiply it by numbers 1 to 12 (yes, I'm old fashioned, why stop at 10?). The loop that starts with:

```
FOR p = 1 TO 12
```

will see to this set of numbers, and the multiplication is done in line 70. It will start with the first of the values that you typed in line 30, and multiply by 1, 2, 3... up to 12. The 'NEXT p' step in line 80 ensures that this loop is completed, because when any loop is started inside another loop, the inner loop must be completed with its NEXT before the outer (first) loop is completed with its NEXT. This is where the word 'nesting' comes from – a loop is completely surrounded by another as a nest surrounds the bird. Poetic, these computer people, aren't they?

We then have to arrange to hold everything. There would be little point in printing one table on top of another so fast that you couldn't read them. The computer normally guards against this by printing 'scroll?' when the screen is full and stopping until you answer with y (to move on) or n (to stop), but if we rely on this it's most likely that the screen will fill up with only part of one table printed. What we can do about it is to put in our own waiting command in the form of line 90. When a key is pressed, the next selected number has its table printed out and at the end of the list you are informed by line 110 that there will be no more tables.

Now just to round it off, take a look at Fig. 4.9. It's a name and phone number program, and it makes use of a string array to hold both the name and the number. Many computers will object if you type a name, then a comma, and a number and enter this as a string, but Spectrum doesn't object, so you can type:

```
Smith, D, 0244 51716
```

and INPUT this quite happily as long as the total number of characters in the string does not exceed 30 (Why 30?). You have to type the surname first, with a capital letter so that the recognition part of the program works correctly, and the program starts off with a query on the number of items so that dimensioning can be done.

As for the rest – for most of it you will recognise the steps, but you might like to note how we check for capital letters in line 200 by the fact that they have ASCII codes between 65 and 90, and how we select the last four characters of a string for the number selection. This is done in lines 320 and 325. Normally, you would find the last four characters of a string variable by finding its length, subtracting four, and then using this in a slicing instruction like a\$(LEN a\$ - 4 TO). When we use string arrays rather than plain string variables, though, each item in an array is padded out with blanks to the dimensioned length – in this example, each string will be 30 characters long, and the last four characters will very often be blanks. The loop in line 320 counts the number of these blanks by setting out to count for as long as the character it slices from the end is a blank. The value of x when line 320 is finished is the number of characters in the real string, with all blanks ignored. This number is then used in line 325 to assign a new string, equal to the string array value with no blanks. Finally we slice this to find the last four

```

10 CLS : PRINT TAB 9;"NAME AND
NUMBER"
20 PRINT "Please select by typ-
ing the ":"number"
30 PRINT "1. Enter names.";"2.
Find names starting with a lett-
er.";"3. Find name for a number."
40 INPUT s: LET s=INT s: IF s<
1 OR s>3 THEN PRINT "Mistake_1 t-
o 3 only_try again": GO TO 40
50 GO TO s*100
60 PRINT "Would you like to re-
turn to menu?" "Please answer y o-
r n"
70 INPUT a$: IF a$="y" THEN GO
TO 20
80 GO TO 999
100 CLS : PRINT "Number of items
s,please.": INPUT n: LET n=INT n
: IF n<1 OR n>50 THEN PRINT "Out-
side my limits_1 to 50,please":
GO TO 100
110 DIM t$(n,30): PRINT "Type s-
username first,with CAPITAL": "as ?
first letter.": FOR j=1 TO n
120 PRINT "Item_";j;" is?": INPUT
"";t$(j): PRINT t$(j)
130 NEXT j: PRINT "End of list-
"
140 GO TO 60
200 CLS : PRINT "Please type th-
e letter you want": "and remember
that names start": "with a capit-
al letter.": INPUT c$: IF CODE c$<
65 OR CODE c$>90 THEN PRINT "Y-
ou forgot,didn't you?": GO TO 20
210 IF n=0 THEN PRINT "No names
entered!": GO TO 60
220 FOR j=1 TO n: IF c$=t$(j)(1)
THEN PRINT "t$(j)
230 NEXT j: PRINT "End of entr-
ies starting with ";c$: GO TO 60
300 CLS : PRINT "Please type th-
e last four digits": "of the numb-
er you want.": INPUT n$: IF LEN
n$>4 THEN PRINT "Four digits onl-
y,please.": GO TO 300
310 IF n=0 THEN PRINT "No numbe-
rs were entered!": GO TO 60
320 FOR j=1 TO n: FOR x=30 TO 1
STEP -1: IF t$(j)(x)=" " THEN N
EXT x
325 LET q$=t$(j)( TO x): LET l=
LEN q$: IF q$(l-3 TO l)=n$ THEN P
RINT "t$(j)
330 NEXT j: PRINT "That's it!" :
GO TO 60

```

Fig. 4.9. The name and phone number program.

characters. Take a long close look at this procedure, because it's a very useful technique to use on the Spectrum, and one which you may very well want to use in your own programs.

Note that this program is useful only for as long as you have the computer switched on. When you switch it off, you lose the stored strings, so that we shall have to consider in the next chapter how we can save such information on a cassette for later use. For saving to Microdrive, see Chapter 23.

Chapter Five

Handling Data

Once we start to use arrays of strings and numbers, we are dealing with what is called *data processing* – meaning that the computer is working with lots of information. The data part of it is just whatever the computer is fed with, names, dates, phone numbers, football results, and so on. The processing is what the computer can be instructed to do with the information, like sorting it into order, picking out names starting with a selected letter, looking for a given number, adding salaries, deducting tax, calculating odds, and all the rest. Serious data processing needs a fairly large memory size and some way of sorting the data outside the computer, like cassettes or the Microdrive, and Spectrum, even in its minimum size, fits the bill for a lot of household data processing requirements. The 48K Spectrum with Microdrives would carry out data processing for a small business. This is a *real* computer, not one of these glorified games machines with a tiny memory that needs expensive cartridges added to make it work.

A lot of data processing programs make use of a ‘menu’, a set of choices from which you have to pick one item at a time to use. The program of Fig. 4.9 had such a menu, and we saw there how we could use the number that was typed in reply to the list of options to go to the correct line of the program, by using GOTO s* 100. The trouble with using GOTO in this way is that we have to remember which line to return to and a more useful way of selecting a set of lines is the use of GOSUB, short for GO TO SUBROUTINE. A subroutine is a piece of program that may have to be performed more than once in the course of a main program, and which is set into action (or ‘called’) by the instruction GOSUB followed by the line number at which the subroutine starts. The instruction GOSUB 1000, for example, will call a subroutine whose first line is 1000, and the lines from 1000 onwards will then be carried out until a line containing RETURN is found. When the computer finds RETURN, it will go back to the instruction that follows the GOSUB 1000 which started the subroutine off. If you see GOSUB 1000 several times in the course of a program, then the RETURN will always be to the instruction following the correct GOSUB 1000 – the computer keeps a note of which one has been used. A special piece of memory, called the *stack*, is used for this purpose.

Subroutines are useful because:

- (1) They can be called from any part of your program, and will return to the correct place,
- (2) one subroutine can call another,

- (3) they save having to type the same instructions several times,
- (4) they can greatly assist in planning a program – more of that in the next chapter.

A menu should therefore preferably be followed by a line which uses something like GOSUB 1000*s, so that a different subroutine is called up for each different value of s. It makes using the program easier if the menu selection number, stored as variable s in our example, is entered by means of INKEY\$ rather than by using INPUT, because INPUT requires the use of the ENTER key, whereas INKEY\$ needs only the number key pressed. Remember, though, that INKEY\$ returns with a *string* and it will have to be converted to number form by using VAL before being used to select the GOSUB. The sort of programming used for this is shown in Fig. 5.1. We've assumed a range of 1 to 9 – most menus will have a smaller list of

```

100 PRINT "Select by pressing a
umber key."
110 LET a$=INKEY$: IF a$="" OR
a$=CHR$(13) THEN GO TO 110
120 LET s=VAL a$: IF s<1 OR s>1
0 THEN PRINT "OUT OF RANGE.": GO
TO 100
130 GO SUB s*1000
140 GO TO 9999
1000 PRINT "1000": RETURN : REM
Test
2000 PRINT "2000": RETURN : REM
Test

```

Fig. 5.1. A menu which selects subroutines.

choices, but if more than 9 choices are to be used, then INKEY\$ is not so useful because it detects one key only – you can't enter 10 with the piece of program shown in Fig. 5.1. Your alternatives are then to use INPUT, or to split up the choice into two menus, each with 9 (or fewer) choices. Note, by the way, that line 110 in Fig. 5.1 also tests for a\$ = CHR\$(13). This tests for the ENTER key, whose code is 13, being pressed, because it's possible that you might still have your finger on the ENTER key after typing RUN and pressing ENTER when the INKEY\$ piece of program is used. Testing for this will avoid the program making a false selection if this key is pressed.

Data filing

Any program that works with more than a fistful of data needs some way of storing the data when the computer is not switched on. Even the shortest computer list of names and phone numbers is rather pointless if it has to be typed in again each time the program is used, so that storing data on cassettes or disks, called 'data filing' is an essential part of any data processing system.

Now by this time, you should have a reasonably clear idea about how we can create a file of items by entering them into an array, string or number, but so far we haven't looked at how these can be saved on tape. This process was not available

on the ZX80 or ZX81, so that Spectrum owners who have graduated from these machines will find that this is new work. If, on the other hand, you have never owned a computer before, it will certainly be new work, and if you have graduated from another type of computer or have given up waiting for one from the 'other channel' then you may know about data filing but not how it is carried out on the Spectrum.

One point that will be familiar to ZX users but not to other readers, is that when you RUN a Spectrum program and then SAVE it, you also save the values of the variables that are used. If you save a program in which the variable z is allocated the value 22 and b\$ is allocated to "Name", then these values are saved along with the program when you carry out SAVE "Index", for example. When you switch on again and type LOAD "Index", then play the tape, your program loads (you DID remember about that EAR plug, I hope?) and the values of variables are also loaded, so that if you type PRINT z,b\$, you will get the values 22 and Name printed. Using RUN will clear these values, but if you start your program by using GOTO line number instead of RUN, the program will make use of the values. You can also save a program so that it starts on a specified line – the details are in the manual – which is useful if you want to be able to stop a game half-way through and then take it up again where you left off. It's rather more advanced programming than we want to look at in this section, though.

We need more than this variable saving facility for data-processing (DP) activities, however, because most DP programs will make use of arrays of strings and/or numbers, and we will want to create these strings, save them on tape, replay them from tape, correct them, add to them, delete items and so on. In addition, if we have a program that creates a file, we may very often find that this file provides useful information for another program. A file created by a program which has an input of names, addresses and telephone numbers could also be used by a program which prints messages for Christmas cards, and addresses the envelopes – the ZX Printer sheets can be cut and glued to cards and envelopes, and it is also possible to connect the Spectrum to printers such as the Epson MX-80 which will print on any type of paper (see Chapter 26).

The Spectrum can save the contents of an array by using the SAVE instruction along with DATA. Suppose we have an array of numbers n(1) to n(50), which we want to record on tape. We have to allocate a filename, just as we do for a program; we might call it Ages. The instruction line for saving this file would then be SAVE "Ages" DATA n(). How does the computer find how many items it needs to record? Simple – you have provided this information in the DIM n(50) statement at the start of your program (you did, didn't you?). That's not quite all, though. When the computer comes to the SAVE "Ages" DATA n() part of your program, it will put out the usual message about pressing RECORD and PLAY and then any computer key, and when you press any computer key, it will record the filename, and then the data in separate bursts. You must let it complete this action, and then you can verify that the data has been successfully recorded by using the command:

VERIFY "Ages" DATA n().

Nothing quite beats doing it for yourself, so let's try a simple example. The program of Fig. 5.2 creates a number array from ages of children in a class. The program calls for you to enter the age in years and weeks, and each age is converted into a number of years, with a decimal fraction. When you have finished, and the example is limited to 10 (oh, luxury!) so that it doesn't take too long, the ages are held in an array n which can then be recorded. Try it, using a spare blank cassette to hold the ages data, not the same cassette as you used to

```

10 CLS : PRINT TAB 14;"AGES":
DIM n(10)
20 PRINT "When each number appears, type ""the age of each child in two stages years (ENTER) then weeks"" (ENTER)."
30 PRINT "The maximum number is ten." "The items will be recorded as a file on cassette."
40 PAUSE 250: FOR j=1 TO 10: C
LS : PRINT AT 5,3;"Age ";j;" is"
50 INPUT b,b: LET n(j)=b+.5b+.25
: PRINT n(j): PAUSE 25
60 NEXT j: CLS : PRINT TAB 10;
"List of ages": FOR j=1 TO 10:
PRINT AT j+1,16;n(j): NEXT j
70 PRINT "Now prepare to record the list." "Make sure that you have a cassette in the recorder"
80 SAVE "Ages" DATA n()
90 CLS : PRINT "All finished now verify." "Run the cassette back to its start": PAUSE 250: PR
INT "When you find the start, press PLAY."
100 VERIFY "Ages" DATA n()
120 GO TO 9999

```

Fig. 5.2. A data filing program, using children's ages.

record the program. The program uses familiar instructions, but note lines 40 and 50 which include the PAUSE instruction. PAUSE, as you might expect, causes a time delay, and is used in this program to give you time to look at what has been input. The number following PAUSE determines for how long the pause will be - a figure of 50 corresponds to about a second. The reason for this seemingly odd number is that the TV receivers in the UK display pictures (technically picture fields) at a rate of 50 per second, and Spectrum counts these for its PAUSE operation. In the US, the rate is 60 per second, so you'll need PAUSE 60 for a 1 second delay over there. (In fact each of these scans only produces half the 'lines' of each picture, but Spectrum doesn't mind.) When you RUN and use this program, you should end up with a cassette which contains a file of ages, all verified by the instruction in line 110. Now what do we do with it? We can load such a file into another program by using:

LOAD "Ages" DATA n()

and it will load the array in - but only if the program has prepared space for it by using DIM n(10) - remember that there are ten items on our file.

Suppose, keeping to this simple example, that we want to find the average age of the children, using a separate program. Such a program could take the form which is shown in Fig. 5.3, perhaps. The file of ages is read in, each item added to a cumulative total, then the average, which is the total divided by the number of items (we chose 10) calculated and printed. It's simple (and not strictly necessary, because we could have easily done the calculations as the ages were being entered) but it illustrates how we can make use of filing.

```

10 CLS : PRINT TAB 10;"Average
Ages" "Please prepare data cassette" "to replay the data." "PRESS ANY KEY WHEN READY"
20 LET k$=INKEY$: IF k$="" OR
k$=CHR$ 13 THEN GO TO 20
30 PRINT "LOADING NOW": LOAD "Ages" DATA n()
40 LET total=0: FOR j=1 TO 10:
LET total=total+n(j): NEXT j
50 LET av=total/10
60 CLS : PRINT AT 10,1;"Average
age is ";av;" years."

```

Fig. 5.3. Using the data created in the previous program.

We can also record string arrays using the same type of command, but with a string array name:

SAVE "Names" DATA a\$()

but there is another point to watch for here. When this array is loaded into another program, it will delete any existing array called a\$, as you might expect, but it will also delete any simple string called a\$. Normally, we can expect to use a\$ and a\$(1) as quite separate items, but in this instance it's not possible. Be careful - it pays to keep a list of the variables and the coded names when you design a program so as to avoid this kind of confusion.

An example of string array filing is obviously called for. The program in Fig. 5.4 allows you to create a file of names and of interests. These are put into two separate arrays, but the numbers correspond so that name(1) goes with interest(1), name(2) with interest(2) and so on. The arrays are then saved in line 50 to provide our database. Watch one point about this program - because there are two arrays to save, you will get two separate messages about pressing the PLAY and RECORD keys of the recorder. All you have to do after you have seen the first lot going is to press any computer key when the second message appears, and stop the recorder only when the "All done" appears.

The next thing is to make use of this. If you want to check which of your friends has a particular interest, there is enough information on your tape database to

```

10 CLS : PRINT TAB 6;"NAME/INT
ERESTS LIST": DIM n$(10,20): DIM
i$(10,20)
20 PRINT "Please enter name an
d main interest as prompted i
n the program.";"These will be
saved on cassette.";"PAUSE 50
30 CLS : FOR j=1 TO 10: PRINT
"Name_":; INPUT n$(j):; PRINT n
$(j): PRINT "Interest_":; INPUT
i$(j):; PRINT i$(j): NEXT j: PA
USE 50
40 CLS : PRINT "The arrays wil
l now be recorded.";"Please prepa
re the cassette.";"PAUSE 150
50 SAVE "Names" DATA n$(): PRI
NT "...KEEP GOING...": SAVE "Int
erests" DATA i$()
60 PRINT "All done"

```

Fig. 5.4. Creating a file of names and interests.

allow you to do this. A piece of program like the one in Fig. 5.5 would suffice – try it! The 'special interest' that you are looking for is entered in line 30 as s\$, and then the data cassette is replayed – as before, it's important to leave the cassette playing until the final message of line 60 appears. A loop then compares each 'interest' item with the s\$ that you have typed, and prints the name or names, if any, corresponding to that interest. One minor point in this program is the use of a counter variable, z. This is set to zero in line 30, but is *incremented* (has 1 added to it) in line 60 each time a name is found. In this way, the number of names can be counted to make the final message more useful.

```

10 CLS : PRINT TAB 9;"Who does
what?": DIM n$(10,20): DIM i$(1
0,20)
20 PRINT "Please enter an inte
rest_;" "the computer will find t
he _" "name or names of friends w
ith_;" "the same interests."
30 PRINT "Interest is_":;
INPUT s$(): PRINT s$: LET z=0
40 PRINT "Now prepare to repla
y the data" "cassette.";"PAUSE 2
50: PRINT "Press PLAY now."
50 LOAD "Names" DATA n$(): PRI
NT "KEEP GOING": LOAD "Interests"
" DATA i$()
60 PRINT "All done_stop the re
corder.": FOR j=1 TO 10: IF i$(j)
" TO LEN s$)=s$ THEN PRINT "n$(j
)": LET z=z+1
70 NEXT j: PRINT "z;" names a
bove are interested in ";s$"

```

Fig. 5.5. A program which uses the data from the previous program.

There's just one snag here. In line 60, we've set up the loop FOR j=1 TO 10, knowing that there were 10 items on the tape. When a tape is created by a program

```

10 CLS : PRINT TAB 12;"DATABAS
E"
20 PRINT "TAB 14;"MENU"
30 PRINT "1. Create List.";"2.
Record List.";"3. Replay List."
;"4. Find an Item."
40 PRINT "Please select by nu
mber"
50 LET k$=INKEY$: IF k$="" OR
k$=CHR$ 13 THEN GO TO 50
60 PAUSE 25: LET s=VAL k$: IF
s<1 OR s>4 THEN PRINT "Incorrect
-1 to 4 only-please try again":
GO TO 40
70 GO SUB 1000+s
80 CLS : PRINT "Would you like
to return to the" "MENU now (y/n)?""
90 LET k$=INKEY$: IF k$="" OR
k$=CHR$ 13 THEN GO TO 90
100 IF k$="y" OR k$="Y" THEN GO
TO 20
110 GO TO 9999
1000 REM Put list creating routi
ne here
1050 PRINT "Create list": PAUSE
150
1100 RETURN
2000 REM Put recording routine h
ere
2050 PRINT "Record list": PAUSE
150
2100 RETURN
3000 REM Put replay routine here
3050 PRINT "Replay list": PAUSE
150
3100 RETURN
4000 REM Put item finder routine
here
4050 PRINT "Item finder": PAUSE
150
4100 RETURN

```

Fig. 5.6. A skeleton data-processing program.

which then uses the same tape, this presents no difficulties, because even if the number of items is stored as a variable, the value can be saved as the program is saved. What if the data is used by another program, however, which has no indication of the number of items? The answer to this one is to have the number saved separately, and the way for doing this which is provided by Spectrum is either to make the number an array with just one item and save it as an array, or to have a separate line, say line 1, which contains the value of the number, saved before the data list is saved. By using the MERGE command, this line 1 can be added to the program which uses the data, and the number taken from it by the program. This, once again, is rather more advanced programming that we want to display in this section. Another problem that you might want to consider is that it would be more convenient to read in at the start of the program, rather than later in line 50. We could then use the information several times rather than having to load it in each time the program is used to find an interest. Once the array has been

recreated, we can use lines like 30 and 60 many times over without loading in more data.

Most data processing programs would be organised with at least four items of menu choice, one being to create the database, one to record it, one to replay it, and one to make use of it. Each choice would be written as a subroutine, and a skeleton outline of such a program is shown in Fig. 5.6. The main 'core' of the program deals with the menu and selection from the menu, with a chance to return to it after a routine has been carried out. All of the actions described in the menu are then carried out by the subroutines, starting at line 1000, 2000, 3000 and 4000. We haven't written these subroutines, because they would take up too much room in an example of this sort, but you have enough experience now to fill your own subroutines. As practise, you can use the pieces of program that we have illustrated in this chapter, but with the line numbers changed to suit the subroutine line numbers that you want to use.

Sorting and selecting

Some of the most useful of subroutines for use in data processing programs are those which deal with sorting and selecting. Sorting means putting into order, alphabetical or numerical, and involves comparing the ASCII codes of items or the number sizes. Selecting means picking out items because of some attribute like number of characters, starting letter, etc.; both need quite a lot of programming. Of the two, selecting routines are simpler, and we'll illustrate two of them here. The subroutine in Fig. 5.7 will find which strings in an array have a given starting letter,

```

999 GO TO 9999: REM avoids crash!
1000 PRINT "Type first letter please": INPUT s$: LET k=0
1005 IF LEN s$ > 1 THEN PRINT "One letter only, please": GO TO 1000
1010 FOR j=1 TO n: REM n is total number
1020 IF s$=l$(j)(1) THEN PRINT "Item ", j, " is ", l$(j): LET k=k+1
1030 NEXT j: IF k=0 THEN PRINT "No matching items"
1040 RETURN

```

Fig. 5.7. A subroutine for selecting strings with a given starting letter – this needs a main program to call it up.

which you are asked to type in line 1000. Note that the subroutine, which is shown as starting in line 1000 (choose your own line numbers, but remember that numbers like 1000, 2000, etc. make it easier to use instructions like GOSUBs* 1000 has a line 999 GOTO 9999. This is put in to prevent the subroutine being used accidentally – what is called 'crashing through'. If your main program finished at line 500, but has no GOTO 9999 line, there's nothing to prevent the computer then moving on to line 1000 and carrying out the subroutine, whether you want it or

not. Mostly you don't so a line like 999 is a way of ensuring that this doesn't happen. You might call it 'defensive programming'.

The rest of the subroutine contains no surprises. There is a mugtrap to guard against two letters being entered, and it could be improved by testing for numbers and perhaps for lower-case letters. Note the use of the counter variable k in lines 1000 and 1020. If no names are found to match your requirements, k will still be zero at the end of the subroutine, and this can be tested and used to print a message, as is illustrated in 1030. There is nothing so frustrating in program terms as a subroutine which does *nothing*; one, for example, which if no match is found, simply returns to the program that called it without printing anything, leaving you wondering if the program is working. By using the counter variable, you can print a message if no names are found, and another which states how many names *have* been found. The value of k may be used by the main program in other ways too (we say the value of k is *passed back* to the main program) such as to pick out k names, using a loop which starts FOR j=1 TO k, and then runs through all the names looking for k of them which have the same starting letter.

Figure 5.8 shows a very different type of subroutine which is very useful in

```

1999 GO TO 9999
2000 PRINT "Type letter group required": INPUT s$: PRINT s$:
LET k=0
2005 LET l=LEN s$: FOR j=1 TO n:
REM n is number of strings.
2010 FOR z=1 TO LEN l$(j)-l+1: z
IF s$=l$(j)(z TO z+l-1) THEN PRINT "Item ", j, " is ", l$(j): LET k=k+1
2015 NEXT z: NEXT j: IF k=0 THEN PRINT "No strings found"
2020 RETURN

```

Fig. 5.8. A subroutine which will find a group of letters in a list of longer strings.

finding names. What it looks for is a group of letters contained inside a string, so that if you know that the item you are looking for contains the word ELECTRONIC or even the part word Elec you can find each string that includes these letters. In this example, if you had decided to look for ELECTRONIC you could pick out strings which contained the words ELECTRONICS, ELECTRONIC ENGINEER and so on; if you had decided to look for Elec, you would also get Electrical Times, Election Special, Electoral liability, Electroluminescence ... and so on.

The routine is not quite as simple as the one in Fig. 5.7. It starts by finding the length of your input string, and the routine would be improved if you set a limit on the length of string (more correctly, substring) that you are looking for, so you might think of a mugtrap for this point. A loop then starts, which will take all the items of the array, one by one, since n is the number of items. The variable n is an example of one which has to be passed to the subroutine. It has been allocated a value in the main program, and is then used in the subroutine. Similarly we can

speak of variables being 'passed back', meaning that they have been allocated values in the subroutine which can then be used by the main program.

Going back to Fig. 5.8, for each string, a second loop starts in line 2010 which selects groups of letters from the string. Suppose that the string which we are checking has 10 characters, and we are looking for a group of 4. We would first check characters 1, 2, 3 and 4 of the string to see if they matched our group of four, and if they did not, we would try characters 2, 3, 4 and 5, then 3, 4, 5 and 6 and so on until we checked 7, 8, 9 and 10. Now character 7, the last starting point for checking, is $10 - 4 + 1$; in terms of variable names, that is $\text{LEN} l$(j) - l + 1$, and that is why this expression appears in line 2010. The string slicing is $z \text{ TO } z + l - 1$ (starting number TO start+length-1), and if the groups of letters match the item is printed. Each item which is found is counted by using $\text{LET } k = k + 1$ as before, and if $k = 0$ at the end of the routine a message to this effect is printed.

Defined functions

When you want to use a subroutine, you have to 'call' it by using the statement **GOSUB** *line number*, with the correct line number for the start of the subroutine. If the subroutine uses a variable *n* which has to be passed to it from the main program, then there must be a value allocated to *n* before the subroutine is called. If the variable that the main routine uses is called *g* and the one that the subroutine uses is *n*, then you must have a line like **LET** *n=g* at some place before the subroutine is called. A *defined function* is another way of calling up an action – but it does not need any line number to be specified, and it is easier to pass quantities to it.

A defined function uses the instruction **DEF FN**, which is obtained (all of it) when you press both shifts, and then release CAPS SHIFT and press 1. The key has **DEF FN** marked below it in red, though the manual suggests that only **DEF** is marked.

The keyword set **DEF FN** has to be followed by a name for the function, which is a single letter, and then by brackets. Inside the brackets are the quantities that the function will need to work on, called its *arguments*, and these will be in the form of variable names *which need not be the names you intend to use*.

Let's see how this might be used, taking as an example a program which used the first letter of a string as a way of selecting items. Somewhere in our program we could have the line:

DEF FN *p\$(q\$)=q\$(1)*

which means that the action called **FN** *p\$* will take the first letter of a string which for convenience only we refer to as *q\$* (because we have to call it *something*). Now if at some other place in your program you have a line:

PRINT FN *p\$(s\$)*

then what is printed is the first letter of the string called *s\$*. It's just as if you had a subroutine which selected the first letter of a string called *q\$*, but which also

included the line **LET** *q\$=s\$*. The variable name which you enclose in brackets after **FN** *p\$* (or whatever you call it) is passed to the function and used in the way that the 'dummy' names (like *q\$*) were used when the function was defined. The important point is that the names of these variables need not be the same. This gives us a lot more freedom, particularly because we can place **DEF FN** anywhere we like in a program, and the computer will ignore it until it is called, unlike a subroutine.

Not many computers permit this very useful method of carrying out operations on variables, so it's likely to be new to you even if Spectrum is not your first computer. The extra freedom takes some getting used to, so we'll look at a few examples; but these only skim the surface of what can be done using defined functions, so it's important to explore the use of this instruction for yourself.

Figure 5.9 is an example of a defined function used with number variables in

```
10 DEF FN c(s)=INT (s*100+.5) /
100
20 LET k=156.274: PRINT k: PRI
NT FN c(k)
30 LET k=FN c(k): PRINT k
40 LET k=200.158
50 PRINT k: PRINT FN c(k)
```

Fig. 5.9. A defined function for rounding off money sums.

money-handling programs – it rounds off a sum to the nearest penny when operations like taking 15% VAT produce fractions of a penny. If we have a sum of money stored as a value for the variable *k*, then we can perform the rounding by using **FN** *c(k)*. In the example, we have printed the value that this produces, but in a real program we might assign this to another variable, or even back to *k* by using **LET** *k=FN c(k)*, so that the rounded value could be used in the next part of the program.

Figure 5.10 illustrates a defined function which uses strings. **FN** *c* this time

```
10 DEF FN c(a$)=16-(LEN a$)/2
20 LET x$="Title"
30 PRINT TAB (FN c(x$)); x$
```

Fig. 5.10. A defined function for centring a title.

returns a number which is the TAB number for printing a string at the centre of a line. It's straightforward and simple to use, which is what defined functions, despite their rather forbidding name, are all about. Note, incidentally, that the computer completely ignores **DEF FN** until it is called for by using **FN**. Figure 5.11 illustrates this by placing **DEF FN** at the start of a line with two other instructions following it. When this is run, the word 'title' is printed, showing that the last statement in line 10 has been carried out, and the words 'Look here' are printed underneath, showing that line 30 has been carried out using the defined function. The word 'TITLE' is not printed, because *a\$* is not changed when line 10 is

```

8 LET a$="title"
10 DEF FN h$(a$)="LOOK here":
LET x=LEN a$; PRINT TAB x;a$;
20 LET a$="TITLE"
30 PRINT FN h$(a$)

```

Fig. 5.11. Illustrating how the computer ignores DEF FN until it is called.

executed, and the a\$ that is used in the function is NOT the same as the a\$ that is used in the rest of the program.

When a variable name is used in the brackets of a DEF FN instruction, like a\$ in Fig. 5.11 or s in Fig. 5.9, that variable is *local*. It is used within the defined function, but is kept separate from any variable of the same name that is used in any other part of your program. You can use the variables a\$ and s in your main program, and the values that they have in the program will not affect the defined function, nor will the action of the defined function affect the variable values in the main program unless you deliberately program them that way. This is a relief, because the defined function allows only single-letter variable names, and you might think you were in danger of running out of names if you made a lot of use of defined functions.

One final point is that the defined function has a limitation – it must be capable of being written in the form DEF FN x(y)=something, and the 'something' has to make sense, it must be something that could normally be placed after equality (assignment) signs. Anything that could follow LET x= or LET x\$= will be acceptable as following DEF FN x(y) =, but items like multistatement lines cannot be used in this way.

Chapter Six

Do-it-yourself Programming

Using your Spectrum to enter programs that you find in magazines or in books can give you some feeling of accomplishment. Running programs, whether for educational, business or leisure uses can bring a lot of satisfaction. For sheer joy, however, nothing beats devising your own programs, entering them, sorting them out, and running them successfully. One authority says that it beats sport, chess or sex. I think it certainly beats sport, and I was never all that keen on chess, but I wouldn't make exaggerated claims. This chapter will concentrate on some of the essentials of how to write your own programs in BASIC.

Now it's always difficult to give advice to people whose requirements may be very different. Reader A wants to write a program to catalogue his stamp collection, reader B likes arcade games, reader C is fascinated by programs that write poetry, reader D wants to keep accounts for tax purposes, reader E likes adventure games, and so on. Everybody who buys a Spectrum has one main reason for wanting to use a computer, even if it's only to keep up with the rapid advances that have been made in this topic, and by the time you have reached this chapter, you probably have some idea about what type of program might be most useful to you. We haven't covered graphics or sound yet, but this looks like a good place to put in some advice about writing programs – a really complete study of this topic would take another book.

We'll start with two points that are more important than practically all of the other advice that you'll read. Point one is that experience counts, and you learn more, and quicker, by trying out ideas on the computer than by all the reading you can ever do. If you've successfully written one program that is original (as far as you are concerned) then even if it is very simple you will have learned more about programming and about your computer than someone who has read a dozen books but never written a program. Point two is that designing a program should be done as far away from the computer as you can get! Why? If you sit at the computer and try to design a program, you'll end up with a Frankenstein program, all bits and pieces bolted together (usually with GOTOS). A computer is an irresistible temptation – if you sit near the Spectrum you are sure to want to switch it on, and when it's switched on you will want to press keys. Design your programs out of sight of the computer, and the temptation becomes less, with the result that the programs become better.

Down to business now. Let's look at how we design two types of programs, one which depends on a menu (a 'menu-driven' program), and one which goes from one step to another as so many adventure games do. With these two types of

program as examples, you should then have a fair idea about how to start designing a program to suit your own needs. We'll start with the menu-driven program, because it's simpler.

To design a program of either type, you need lots of paper. Buy scrap-pads if you like, but I personally prefer to work with ruled A4 sheets which I can then file into folders. That way, all of my ideas on a program can be kept together, but I can throw away any old or half-baked stuff rather than have it cluttering up the folder. The first step is to write down what you want the program to do. You may think that this is unnecessary, but you will be surprised to find how quickly the program that you are writing starts to differ from the program you intended to write. If you put down clearly on paper what you intend your program to do, you can keep checking as you design the program that your specification is being met. It's also a good idea to keep this specification up to date, if you have any second (third, fourth ...) thoughts as you go about writing the program.

The next step in a menu-driven program is to decide what the items of the menu should be. You will usually need to have choices such as *create a list, add items, delete items, save list on tape, replay list from tape ...* in any menu-driven program, so that you should be able to design your menu by thinking of what your program is supposed to do – and that's one of the reasons for writing down a specification to start with. You can now plan the 'core' of your program. This should consist of printing a title, possibly some instructions, then the menu list, and finally a return-to-menu option. That's all. If the instructions are long, it's usually better to print brief notes and ask if more is needed (y or n). That way, you don't have a long set of instructions put on the screen time after time, only when you need them, which will be when you haven't used the program for some time. By now, we can draw up a plan for this core section; it might look like the illustration in Fig. 6.1.

CLEAR ➤ INSTRUCTIONS ➤ MENU ➤ CHOICE ➤ EXECUTE ➤ RETURN
SCREEN

Fig. 6.1. A plan for a program – start with the important steps, and don't worry about the details.

The next thing is to work on the core *by itself*. Don't worry at this stage about how you are going to program the actions that take place in the menu options – we'll deal with them later (perhaps we could call this approach *mañana* programming). Right now, the main thing is to get this core section working correctly, which means good presentation and menu selection. You will want to print the title centred against a blank screen, so that underneath the word *title* on your core plan you can write brief notes to remind you that you want to clear the screen and use a defined function to centre the title – after all you will probably want to centre more than one title. You might also like to think about background (paper) and text (ink) colours, even if these are only black and white – there will be more about the use of colour in Part 2.

Moving on to the Instructions heading, you will want to remind yourself that

this would be an option with a y/n choice, and you need to make a note also about how much detail you need. *Don't* write any instructions in detail yet – if you do, it's a pound to a penny you'll have to alter them several times before the program is finished. Moving on to the Menu heading, you know now how many choices you need, and you can write down what first line number will be used for each subroutine. Another note here reminds you that you will need a mugtrap to deal with impossible quantities. Finally, there's the return-to-menu option, another y/n choice. Your plan will now look as in Fig. 6.2, with rather more detail displayed.

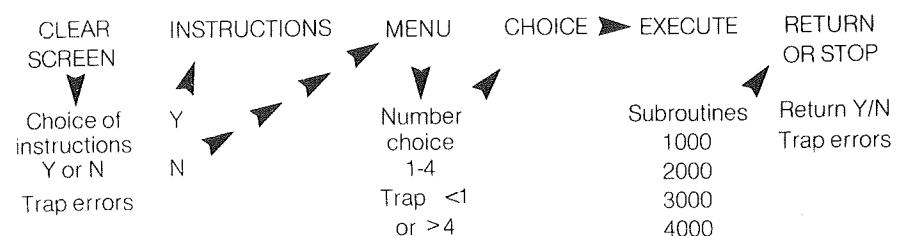


Fig. 6.2. Putting more detail on the plan.

The next step is to design the layout of what you want to see on the screen at each stage and, if you plan to use sound, what you expect to hear. Write down the title you intend to use. Is it reasonably short? If it's long, would it look better split into two lines, each centred? Under it (how many lines under it?) you will want some brief instructions (how brief?), and the option to hit keys y or n to bring up detailed instructions or to go on the the program. This will also have the effect of keeping the title and the brief instructions on the screen until you have read them – if you don't use a y/n at this stage you need an INKEY\$ or at least a PAUSE to give the user time to read the words on the screen. The detailed instructions can be printed by a subroutine which will be called only if there is a 'Y' answer at this stage. It can be provisionally numbered 9500, and we don't need to write it yet.

The next section is the display of the menu, which should start with clearing the screen to remove the title and brief instructions, then displaying the menu choices. Under the menu choices, you need to indicate what the user is supposed to do, such as – *Choose by pressing number key* – if you are using INKEY\$, or – *Press number key, then ENTER* – if you are using INPUT. When this is done, you need a mugtrap and a possible return to menu to detect numbers that are out of range, but no more new printing is needed in the core program until the subroutine is completed. The next step in the core program is then to clear the screen and print somewhere in the middle the words: *Do you want to return to the menu?* (y/n).

By planning your printing, preferably on squared paper with 32 columns and 22 lines, or by using tracing paper over a grid drawn to this pattern, you can see how well words fit on each line, and when you will have to move down a line, using the apostrophe instruction symbol. This can save a lot of time and effort when it comes to programming, so don't stint your planning. Work in pencil, so that you can easily rub it out and try again.

Now you can start programming the core section. It will end up looking very much like the example in Fig. 5.6, because the core of one menu-driven program looks pretty much like any other. The important point is that it is self-contained, with a beginning and an end, and short enough to be easily tested. A very important point to think about at this stage is how the program is affected when something silly is entered. Obviously, *you* wouldn't do such a thing, but fingers brushing against keys can cause odd effects, and a good program should contain enough mugtraps to cope with such eventualities. You should also start to think at this stage about what other subroutines you will need, because there will be some actions that will be carried out several times over, like screen displays, yes/no selection, pauses, and so on.

When you have written the core section, which may consist of only seven or eight lines, enter it into the machine and test it. Before you start to test it, you will have to put some subroutine lines in, with a RETURN in each of them, to make sure that the program does not halt with an error message each time you test the menu choice. If you are fussy, you can put a temporary title to be printed on each subroutine, then a pause, before the RETURN, so that when you test each menu choice, you can be certain that the correct subroutine has been chosen because its title is displayed for a moment before the screen is cleared and the *return to menu* choice is offered. Make your testing thorough, trying each menu number, and also testing your mugtraps by entering numbers outside the menu range, entering letters, and other 'sillies'. Look for poor displays, titles which are not visible long enough to read, and other display errors. The more thoroughly that you can test the core the better, because once it has passed all this testing you know that you will be able to rely on it. When you are satisfied with the core program, record it twice, make sure that you have a note of it as it now exists and if you have a printer, print it. I personally would never attempt to design a program of more than a few lines without the help of a printer, because I can see flaws in printed listings much more readily than on the screen. Once this is all done, switch off the computer and go back to the paperwork.

You now need to design each subroutine. The point now is that you have *no new skills to learn about design!* You can design each subroutine just in the same way as you have designed the core program, by drawing up a list of the main steps, and then filling in reminders about the details, which can be done by other subroutines if needed. Some of your subroutines may be much larger than the core program, so that they need to be broken down in several layers of subroutines, though not necessarily with a menu choice. Other subroutines may be very short, such as the y/n choice, but they still need to be designed, because they will be used many times in the course of a program. As each subroutine is designed, it can be programmed, and then tested. To test a subroutine, you should load in the core program, type the new lines of subroutine, and then start testing by calling the subroutine from the main program – this is particularly important if the main program passes variables to the subroutine. When each subroutine has been tested, save the complete program as it is, and load this version in again when you are ready to type another subroutine. In this way, your program grows steadily, and since each

section has been tested you can rely on each section to work – if something goes wrong it will either be because of a fault in a new piece of program or because a variable name has not been passed correctly or possibly because the same variable name has been used for different jobs. You did keep a note of how you used your variable names, didn't you?

If you design a lot of rather similar programs, it's a good idea to keep a 'library' of subroutines on a cassette. By using the MERGE command of the Spectrum, you can add these subroutine lines to a core program (provided that the line numbers are not also used by the core program) easily, with no more typing. The only changes you will have to make will be to variable names, unless you have been very strong willed and have used consistent variable names in all of your programs. The main problem of using subroutine libraries is that the subroutines may have conflicting line numbers – it may be necessary to renumber some of them.

Screen-driven programs

Most data-processing programs are menu-driven, and they are so similar that they can all be designed in the same way – this is the thinking behind the idea of 'programs that write programs'. Games programs are sometimes menu-driven, but many are screen-driven, meaning that you have to use the keys to respond to or control what is happening on the screen. It's important to realise at this point that the programming language that is used for the Spectrum, BASIC, is comparatively slow, too slow to allow fast animated games to be programmed in BASIC. The Spectrum BASIC is slower than that of some other computers, and for arcade-type games it is essential to use a different language, one that will issue instructions direct to the microprocessor chip which controls all the actions of the Spectrum. This language is called Z-80 Machine Code and it is a specialised subject dealt with in Part 6.

What sort of games are open to us, programming in BASIC? We can design games of options, adventure style, games of graphics (but not if rapid animation is needed), or games of guessing and decision. Some of these types can involve menus, but the graphics games may require the use of keys to place shapes on the screen, with the keys taking the place of a menu for making choices. The most obvious scheme of this type is the 'sketch' type of program in which four keys are used to control a cursor or other marker to move up, down, left or right, so that lines can be drawn. It should also be possible to turn the cursor on or off, and to delete lines, and to save a drawing on tape – the Spectrum provides commands which make all of these requirements very easy.

The design of a program of this type must start as usual with a clear description of what you want to do. If points are to be scored in the game, you must decide on what basis – making up the rules of a game is by far the most difficult part of all, which is why there are several thousand versions of old games like Hangman.

Having decided what the game consists of, its rules, and how it is to be played, you can then start by drawing up a program plan which might look something like

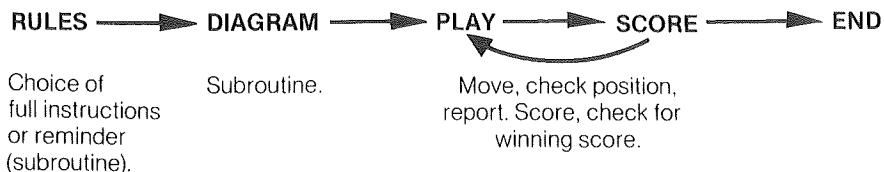


Fig. 6.3. A program plan for a game.

the outline in Fig. 6.3. As usual, there is a title, and there will have to be rules or instructions, which might have to be presented again, or changed, during the course of the game. After that, though, the next heading is *Diagram*, because you may want the screen to show the control panel of the X-fighter, a Map of the Valley of the Giants, a Guide to Treasure Island, or whatever. This will need a subroutine, because you may need to replace this several times in the course of the program.

The next heading is *Play* – you will already have decided what you want to use the keyboard for, and you now have to decide details. Which key switches on the main rocket motors? Which key causes retro-thrust? Which key indicates that a well should be sunk at the cursor position? Which key brings up the next magic weapon? Personally, I think that the programming of such a game is a lot more interesting than playing the game!

You then have to attend to *scoring*. What variable name will be used to hold your score? What variable name holds your opponent's score? Is there a real opponent, or do you play against the computer itself? How many points are awarded for a hit, and how many for a near miss? Are there any fatal events (*You are destroyed by a Gorgon bomb – go back 3 Aeons to be reborn*)? When a score is made, how does the game continue?

As before, this type of outline should enable you to start putting some flesh on the skeleton in the form of details, and eventually you will be able to start writing a core program and deciding on the subroutines you will need. Once again, all the details are contained in subroutines, though in this type of program they are more likely to be set into action by pressing keys than by a menu choice.

At this stage, because we haven't yet dealt with graphics, colour or sound, it's not easy to go into details of programming, but a few tips can be useful. One point is how keys can be made to carry out different cursor actions, without departing too much from simple BASIC, and this can be solved by the use of INKEY\$ as shown in Fig. 6.4. When a key is pressed, it is tested to find if it is one of the five

```

100 LET x=15: LET y=11
110 LET k$=INKEY$
120 IF k$="u" THEN LET y=y-1
130 IF k$="d" THEN LET y=y+1
140 IF k$="l" THEN LET x=x-1
150 IF k$="r" THEN LET x=x+1
160 IF k$="a" THEN GO TO 180
170 PRINT AT y,x;"."; GO TO 110
180 REM Next instruction
  
```

Fig. 6.4. Using keys to shift a print position – the basis for a lot of action games.

keys listed. Why five? We need four movement keys, up, down, left, and right, and one to escape from the loop (a for out), because this is a closed loop which will return to 110 until the a key is pressed. Since the loop will repeat its action for as long as a key is pressed, it will shift the printing of the dot (in this example) in the direction indicated by making use of the changes in the x and y PRINT AT numbers. This is the basis of the sketch type of program, though it can be done much better by using the PLOT command (see Part 2), and with provisions for rubbing out the trace, or for leaving no trace.

Another problem concerns selecting an item at random. There are several ways of doing this, of which the easiest are the use of string arrays and of the RESTORE instruction. The use of a string array is illustrated in Fig. 6.5. The items are read into an array, l\$(n), and a number found at random. The item of the list corresponding to this random number is then used in the program. The other option consists of putting each item into a DATA line of its own (Fig. 6.6) and using the random number to select the correct data line by RESTORE 10*k or

```

100 DIM l$(n,10)
110 FOR j=1 TO n: READ l$(j): N
EXT
120 LET k=1+INT (RND*n): PRINT
"Pick up a "; l$(k); " and use it!"
  
```

Fig. 6.5. Using a string array from which items can be selected (no DATA line shown – use your imagination!).

```

10 DATA "giant"
20 DATA "troll"
30 DATA "venusian cephalopod"
40 DATA "shop steward"
50 DATA "vampire"
100 LET k=1+INT (RND*n): RESTOR
E 10*k
110 READ l$: PRINT "You are men
aced by a "; TAB 10; l$
  
```

Fig. 6.6. Putting each item in its own data line and using RESTORE for selection.

RESTORE 900+k, or similar. This also has the advantage that the program will be peppered with data lines (when the RESTORE is of the form 10*k), so making it more difficult for anyone looking at the listing to discover what is going on!

As an example to build on, Fig. 6.7 shows a core program for a simple game based on animals and their young, aimed at young computer users (there's no special name for them, so far). If you are old enough to regard this as kids' stuff, don't neglect the ideas because they can be used as the basis of something which you might find a lot more appealing to your own taste.

As it appears in Fig. 6.7, the game is very simple indeed, calling for an answer which is compared with the list of answers held in a DATA statement, and then indicating correct or otherwise, and showing a score. What makes it more interesting is that it is easy to expand. As it stands, an animal name is picked at random by using a random number to select one item from a string array, a\$(n).

The same number can then be used to select the correct name of the young from the other string array y\$(n). The answer given by the user is compared with the answer held in the string, and if they are identical, the score is increased by 1 and the score is printed. A point to remember here is that the input string x\$ is NEVER identical to the answer string, because the answer is in an array, and its string contains blanks to make the length equal to the dimensioned length. This is the reason for the expression in line 100 which selects as many characters from y\$ as there are in the string x\$ before comparing. Watch for this point if you are converting a game written for another make of computer into Spectrum BASIC.

```

10 CLS : PRINT TAB 10;"ANIMAL
GAME" AT 21,1;"@ Ian Sinclair 19
30: PAUSE 150: CLS
20 PRINT TAB 10;"Instructions"
"YOU will be presented with the
""name of an animal. You should
""then type the name of its young"
"and press ENTER.";"The computer
will keep score.": PAUSE 20
@ 
30 DIM a$(10,5): DIM y$(10,7):
REM animals and young
40 FOR j=1 TO 10: READ a$(j):
NEXT j: FOR j=1 TO 10: READ y$(j)
: NEXT j: REM put in arrays
50 LET t=0: LET s=0: LET g=0:
LET b$="Animal": LET z$="Young":
REM tries, score, guesses, titles
60 LET t=t+1
70 RANDOMIZE: LET r=1+INT (RN
D*10): REM pick at random
80 CLS : PRINT AT 5,10;"_";a$(r)
90 PRINT AT 8,1;"What do you call
its ";z$;" ?": INPUT "Please
answer here_";x$:
100 IF x$=y$(r) THEN LEN x$) THE
N LET s=s+1: PRINT "You scored_
total is ";s;" in ";t;" goes.": 
PRINT "-now for another one.": L
ET g=0: PAUSE 100: GO TO 60
110 PRINT "Not correct-but it
might just be your spelling o
r lack of a capital letter"
120 IF g=1 THEN PRINT "No more
attempts allowed.": "We'll try an
other ";b$;": PAUSE 100: LET g
=0: GO TO 60
130 PRINT "Try again-you get th
is one free";"in a moment": LET
g=1
140 PAUSE 150: GO TO 80
150 GO TO 9999
200 DATA "Horse","Cow","Cat","D
og","Goat","Hen","Swan","Goose",
"Wolf","Pig"
300 DATA "Foal","Calf","Kitten",
"PUPPY","Kid","Chicken","Cygnet",
"Gosling","Cub","Piglet"

```

Fig. 6.7. The animal game – showing how such games are constructed.

If the answers do NOT match, then the player gets another chance – the counter g is incremented from 0 to 1. On the second guess, if there is no match and with g=1, no more attempts at the same animal are allowed, and a new name is presented. Because of the way in which random selection works, however, this new selection stands a 1 in 10 chance of being the same animal! You could expand the program to eliminate this possibility by swapping this animal and its young for the last one in the list (for example, if it is No. 5, make it No. 10, and make animal and young No. 10 into No. 5, then alter the random choice to a choice from 9 only). The new instruction RANDOMIZE (RAND on the T key) is used in line 70 because RND used by itself eventually starts to produce repeating patterns, and RANDOMIZE prevents this.

Now let's look further at how we could build on this foundation. One obvious step would be to have very many more pairs of adult/young animals. With the RESTORE line-number type of instruction that Spectrum permits, it would be possible to play the game at different levels of difficulty, with domestic animals for the younger user, and less familiar animals for the older. A line which printed the message 'What level of difficulty would you like? (1 to 3)' followed by selecting a number which restored a different DATA line would carry out this action. You can pack a lot of words on one data line, many more than other computers allow, so that this approach is particularly suited to Spectrum. Another feature for younger users would be to draw the animal piece by piece as a clue, knocking points off as the drawing proceeds, so that 10 is scored for a correct guess made before drawing, and only 1 when the drawing is complete. The drawing would be done by a subroutine which is selected by the random number – but be careful if you are also shifting the animal names around to prevent duplication – you could find yourself with a drawing of a duck when the guess is a dog!

We can make further complications. One obvious one is that the user who knows how to LIST a program can discover the answers, as we have hinted earlier. There are several ways around this, one of which is to tamper with the computer's memory so that LIST has no effect, but we'll stick to BASIC programming in this section. An ingenious solution is to store a string or a DATA line which simply consists of the alphabet. Each answer is made up by selecting letters, using j\$(n) for the string selection, to select a letter which is the *n*th letter of the alphabet. This needs a lot of work – to read FOAL you need the number sequence 6, 15, 1, 12 because these are the numbers of the letters F,O,A,L in the alphabet string (and your answers will now have to be in upper case, unless your alphabet is stored in lower case and you use a routine to make the first letter into upper case by working on the ASCII code). This type of disguise is very effective when the user does not know the code, and it slows down the user who does realise what is being done!

Another possibility is to store the characters of the answer as ASCII codes, or as 500 + the ASCII codes (better), or some other method based on the ASCII codes for the letters. The codes can be stored as DATA numbers or as strings; if they are stored as strings it is an advantage to use something like 500 + code, because each string will then be of the same length, and they can be packed together into one string like 500567543 to be separated later into three-digit groups. The separated

strings will then have to be converted back into numbers before CHR\$ can be used in a loop like:

```
a$="":FOR k=1 TO n: READ x:a$=a$+CHR$(x-50):NEXTk
```

where k is a number that has to be read to indicate the length of the string.

Another possibility is to introduce a choice of games based on the same set of animals. You could think of associated items like female of the spaces, habitat, favourite food ... the possibilities are endless. The program could then start off with a menu which asked which game you wanted to play. The selection which is made at this point would then pick the DATA line that was used for the answers and the name allocated to z\$ in line 90, but very little else.

Whatever your programming interests, any time that you spend designing a program in sections that can be tested separately or successively is well worth while. In this context, remember that if you write a subroutine which uses, say, variables a\$, b\$, c\$, p, q, r, and j, then you *don't* have to attach a core program to it in order to test the subroutine. Simply use direct commands like LET a\$= "Trolls" (press ENTER), LET b\$ = "Amulets" (press ENTER) LET p=666 (press ENTER) and so on until all of the variables have been allocated, then type GOTO followed by the starting line of your subroutine. Don't use RUN, because the RUN command will wipe out all allocations of values to variable names. You can also use direct commands to find out what has happened to variables after a routine has run, by typing PRINT a\$, PRINT p and so on.

Chapter Seven

Other Instructions

Editing means correcting mistakes, and the more easily this can be done, the more useful the computer. To start with, you already know how to delete a mistake by using the CAPS SHIFT and the Ø key (DELETE) together, and this action is the basis of editing. What we need to look at now is how to amend a mistake in a line as it is being typed, when you don't want to have to rub out a lot of characters, and how to amend a mistake in a line that has already been typed and entered. These require the editing steps of Spectrum to carry out.

Deleting or adding a complete line is simple enough. To delete a line, simply type its line number and then press ENTER. To add a new line, type the line number, then the line and ENTER in the usual way. The more awkward problem occurs when you find that you want to alter a word or a letter within a line without having to type the whole line again. This is simple if the line is still in the entry position, on the two bottom lines of the screen. If it is not, then the steps you need to get it there are:

- (1) Press the LIST key, and then type the line number. When you press ENTER, the listing will appear with a cursor arrow pointing to the line you want.
- (2) Alternatively, when you have LISTed, the cursor points to the first line of your program. You can move the cursor down the listing by using the down arrow key (CAPS SHIFT 6) and up the listing by using the editing up-arrow (CAPS SHIFT 7).
- (3) To put the selected line into the bottom lines for editing, press the EDIT key (CAPS SHIFT and 1).

Whether you already have a partly completed line on the bottom line spaces, or have just moved a complete line down to these spaces using the techniques described above, you can now edit this line. The cursor can be moved along this line (or lines) by using the side-arrow keys, obtained with CAPS SHIFT and 5 or 8. The cursor will be at the end of the line, if you are typing a line and want to correct a mistake at the beginning, and it will be placed at the start of the line, between the line number and the first instruction if the line has been brought down to the bottom line. Move the cursor where you want it – the cursor arrow keys have auto-repeat, so that you can keep the cursor moving just by holding the keys down. Moving the cursor in this way does NOT cause any change in the line. To add new statements to a line, place the cursor where you want the new material, and type. You will have to pay attention to the shape of the cursor, just as you do

when you are typing a new line – when it is a K you will enter a keyword, when it is an L you will enter letters, and so on. If you are placing a new statement into the middle of a line you will probably have to start and end with a colon.

To delete an entry, move the cursor to the immediate right of whatever you want to delete, and then press DELETE (CAPS SHIFT and \emptyset). This key also repeats, so you will have to be careful that you do not delete too much by holding the key down. A keyword will always be completely deleted by one press of the delete key, but words in a PRINT statement have to be deleted letter by letter.

When you have finished editing, press the ENTER key, and the new version of the line will replace the old version in the memory. All parts of a line can be edited – and that includes the line number so that if, for example, you want to change line 20 into line 1000 so as to make it into a subroutine, this is simply done. Just bring the line down, delete the line number, type in a new one, and press ENTER. It's beautifully simple compared with the fuss that some machines need, but if you want to renumber a lot of lines in a new sequence, several 'toolkit' programs are available for this (see Appendix A). At the end of an edit, the editing cursor will still be positioned over the line, so that if you have second thoughts, the EDIT (CAPS SHIFT and I) procedure will bring down the line again, with no searching needed.

ZX81 compatibility

ZX81 cassettes cannot be loaded into the Spectrum as it stands, though programs are available which permit this to be done – keep a close lookout for advertisements in the computing magazines. In general, any program that has been written for the ZX81 can be typed into the Spectrum, providing that it is *purely in BASIC*. A few instructions may, however, have to be omitted or modified. In particular, FAST and SLOW instructions in a ZX81 program will have to be removed, along with SCROLL; UNPLOT statements will have to be replaced with PLOT OVER 1 and this may require some changes to what is plotted over.

The programs which cause real difficulty are those which use the operating system of the computer and incorporate PEEK and POKE instructions. A program that makes extensive use of these commands will be very difficult to convert, because it requires good knowledge of the operating systems of both computers, and no beginner to computing is likely to have such knowledge. There are a number of books and articles on the subject, but for the moment, it is better to leave such programs alone. In general, programs which will run on the ZX80 and ZX81 can usually be completely re-written for the Spectrum so that they take full advantage of the unique features of Spectrum, and this is much more useful than simply making minor adaptations.

Some other instructions

The instructions POKE, PEEK and USR are fully dealt with in Part 6, but there

are a few instructions and print marks which we have not covered, some because they are useful only when attachments are connected to the Spectrum, some because they are seldom used, and some because they are so specialised that we need only mention that they exist.

Instructions which fall into the first group are LPRINT, LLIST, which are used by ZX-compatible printers, LINE, OPEN, CLOSE, MOVE, ERASE, CAT and FORMAT, which apply to the Microdrive, and IN, OUT which are used when the Spectrum is linked to other devices such as controllers. For any serious work on writing programs of more than a few lines, a printer is almost essential, so that the use of LPRINT and LLIST may be of interest here; the Microdrive commands are fully dealt with in Part 5, as are the similar commands for 'networking' – linking up to 64 Spectrums together. LPRINT is used in place of PRINT when the words inside quotes are to be sent to the printer rather than to the screen. If LPRINT is used without the ZX printer attached, the instruction will simply be ignored; it does not cause the computer to lock up, as some do. If you want to use a program that contains a number of LPRINT statements, and you have no printer, then simply edit all the LPRINTs into PRINTs. LLIST similarly causes a program listing to appear in print, with no waiting for a scroll instruction.

Of the second group, many of the mathematical instructions (strictly speaking, functions) are of use only to readers who know enough of mathematics, but a few are worth mentioning. ASN, ACS and ATN give, respectively, the angle whose Sine, Cosine, and Tangent value has been entered after the instruction. For example, PRINT ASN .012 will give .01200028, the angle *in radians* (see Part 2) which has a sine value of .012. To convert this angle into degrees, you need to multiply by 180 and divide by PI (above the M key) – note that the angle in radians is almost equal to the sine of the angle, which is another reason for using radians in angle measurement rather than degrees.

The SGN instruction is used to find the sign of a number variable. PRINT SGN A will, for example, give +1 if a is positive, \emptyset if a is zero, and -1 if a is negative – Fig. 7.1 shows the instruction in action, put into a subroutine. ABS is an

```

10 LET a=5
20 GO SUB 100
30 LET a=0
40 GO SUB 100
50 LET a=-5
60 GO SUB 100
70 GO TO 9999
100 PRINT "Sign is ",SGN a
110 RETURN

```

Fig. 7.1. Using SGN to find the sign of a variable value, or if the value is zero.

instruction whose effect is to strip the sign from a number (or a variable used to represent a number). If you have a line which is: 'LET a=-6', then 'PRINT ABS a' will give 6, having removed the negative sign. This is useful if you want, for example, to print squares of numbers (or other powers) because Spectrum refuses to give a value for the square of a negative number, and programming PRINT a²

will come up with Invalid argument if a is negative. You can, however, use PRINT (ABSa)^{1/2}, and this will work. Moving on, the LN instruction gives the natural (Napierian, base) logarithm of the number (or number value or variable) which follows it. This is useful in calculations on capacitor discharge, radioactive decay, emptying of reservoirs, and other natural processes, but if you want ordinary base 10 logarithms (for decibel calculations, for example), you will have to use 2.303*LN a.

The last group consists of the instruction VAL\$, which no-one appears to have found a use for to date, and some of the odd marks like the square and the curly brackets (under keys Y, U, F, G), the sign ~, which is used above some letters (usually n) in Spanish, and also used as a mathematical symbol for 'approximately', and the vertical line and backslash on the S and D keys. Few of these are likely to find their way into our programs.

However, there is one very large group of commands that we have not covered at all in this section – the commands that allow you to use the Spectrum's very useful graphics and sound facilities. These really deserve a section to themselves.

Part 2

Graphics and Sound

by Ian Sinclair and Steve Money

Graphics and Sound

One of the most seductive things about the Spectrum is its ability to create fully detailed colour graphics without using up most of its available memory space in the process. Bearing in mind that a full screen contains 256×176 separate dots, or *pixels*, and that the computer needs to know a fair amount of information about each dot, this may sound impossible. After all, it has to keep track of the colour of 45056 different pixels – whether they are ink or paper colour, and if so which (from a choice of eight), whether or not they are FLASHing, and whether or not they are BRIGHT. That's a great deal of information – more, in fact, than the Spectrum could ever cope with. So it cheats.

To see how, NEW out any program you have in the computer at the moment and LOAD a screen display – any screen display. (If all else fails, use the display at the start of your HORIZONS tape.) Normally if you start 'cold' like this the first display will start loading in black and white – black ink and white paper. Look carefully. The first thing to appear is a horizontal line of black dots near the top of the screen. Then a second line appears, some distance down from it. Eight spaced lines will appear, and then a second line of dots will be printed immediately below the first one at the top of the screen, another below the second, and so on. In all, eight separate lines of dots will appear in each strip until the top third of the screen is full – eight strips of eight dot-lines each. Then the process is repeated for the second and third portions of the screen. Finally the colour appears, in a rather more normal pattern. Starting at the top, it shows as a wide band filling one strip of eight dot-lines at a time from top to bottom of the screen. At the same time, any areas of screen that are FLASHing or BRIGHT will reveal themselves.

All the extra information about these *attributes* – colour, brightness, etc. – is stored in a separate part of the Spectrum's memory from the actual dot pattern; and as you may have guessed already, the Spectrum doesn't actually note the attributes of individual pixels, but just those of each 8×8 block of pixels. Within each 8×8 block there can only be two colours (INK and PAPER colour). This is a bit limiting sometimes, but with a little care and ingenuity it's a limitation you can beat, as this section will show you. The Spectrum comes fully equipped with an armoury of useful commands to help you make the most of its graphics capability. Some of those commands are mathematical functions like SIN and COS that you may only associate with school homework, but here you'll find formulae that use them to draw shapes on screen, shapes you can use (if you wish) for your private version of Space Invaders, for drawing plans and diagrams, or for creating charts, graphs, and displays.

The sound facilities on the Spectrum are not so well developed, but they are there and they are useful. The final chapter in this section shows you how they work, and how you can use them to the best advantage either for music (one part only!) or for sound effects and signals within a program. The Spectrum is actually capable of a great deal more in this field, but only with the help of some new *peripherals* (see Part 5). On the other hand, getting to know its BEEP command is a good preparation for more advanced work later on.

Now read on, and start bringing your Spectrum to life!

Chapter Eight

Screen Displays

Graphics is the word used to describe the characters of a computer which are *shapes* rather than alphabetical, numerical or punctuation marks. The provision of graphics characters allows us to draw diagrams and pictures; it's not just a feature that is useful for games, it is valuable for presenting information in the form of pie charts, bar charts and graphs. Like most computers of modern design, Spectrum allows these graphics characters to be chosen from a ready-made set or to be designed by you for your own purposes; they can be high resolution (fine detail) or low resolution (large blocks), and in colour. We shall be dealing with colour in the next chapter, and concentrating on shapes in this chapter, starting with the low resolution graphics shapes.

LRG

LRG is a convenient abbreviation for Low Resolution Graphics, as distinct from HRG (High Resolution Graphics), and the LRG characters of the Spectrum can be obtained by using the keyboard direct, using the number keys on the top row of the keyboard – but you need to know how to obtain these graphics.

Suppose, for example, that you want to print the character  at the centre of the screen. Now the screen centre is at TAB 15 across and line 10 down (remember that numbering starts with zero). The command that you want is:

PRINT AT 10,15;"

and the PRINT AT 10,15; is the easy part. To get the graphics character, you press CAPS SHIFT and key 9 together, which places the computer into its graphics mode, indicated by the letter G appearing as a cursor in place of K, L, E or C. Once this graphics cursor appears, you can type graphics characters directly on to the screen. To get the character  , you type the 6 key when the G cursor is flashing. This leaves you still in graphics mode, and you have to escape before you can type the final quotes mark. This is done by pressing the CAPS SHIFT and 9 keys again. When the line looks complete (did you remember the first set of quotes?), press ENTER and the graphics character will appear on the screen.

The graphics characters which are illustrated on keys 1 to 8 are only half of the available LRG set. If you press *either* of the SHIFT keys (CAPS SHIFT or SYMBOL SHIFT) along with the graphics character key, you will get the *inverse* character – white in place of black and black in place of white. Figure 8.1 lists the possibilities, along with the CHR\$ numbers which produce the same effect in a

| KEY MARKING | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|-----------------|-----|-----|-----|-----|-----|-----|-----|-----|
| (No CAPS Key) | | | | | | | | |
| No. Code | 129 | 130 | 131 | 132 | 133 | 134 | 135 | 128 |
| (CAPS Key down) | | | | | | | | |
| No. Code | 142 | 141 | 140 | 139 | 138 | 137 | 136 | 143 |

Codes 144–164 are reserved for your own graphics characters

Fig. 8.1. The graphics characters which are available on keys.

PRINT CHR\$ number type of statement. By far the simplest way of producing graphics in LRG is to use the keyboard in the same way as you type ordinary alphabetical characters, using the blank graphic on the 8 key as a spacer.

Now how can we make these graphics characters form a shape on the screen? There are several ways, and we'll start with what could be called 'graphics printing', taking as an example the pattern shown in Fig. 8.2, a basic 'flying saucer'. The first thing to do is to draw this in squared-off form, and the easiest way of doing this is to use graph paper, preferably the millimetre/centimetre square type. If you use a large centimetre square to represent a complete graphics block, you can draw in the outline on the paper, and then shade in the quarter-blocks the way you want them to appear on the screen. The next step is to identify these blocks as characters on the keys.

Tab Numbers 13 15 16 17

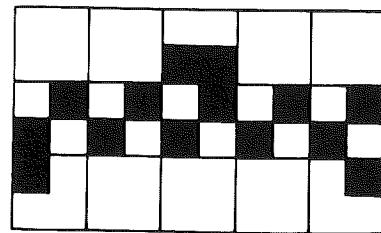


Fig. 8.2. A plan for a 'flying saucer' made from graphics characters.

Using the pattern in Fig. 8.2, if we want this to appear in the middle of the screen line, we shall have to choose TAB numbers to suit the shape, which is 5 characters wide. This will centre reasonably well if we use TAB 13 for the left-hand side, so the next step is to write TAB numbers across the top of the block positions. The pattern uses a depth of three blocks, so three PRINT statements will be needed.

The first line contains the block at TAB 15, so the line to print this is PRINT TAB 15; . That deals with the turret of the flying saucer. The body starts on the next line at TAB 13, and its print line is:

PRINT TAB 13;

The last line is then PRINT TAB 13; TAB 17; , an alternative is to use three spaces between the characters in place of the TAB.

That's it – if you clear the screen and RUN, you should see your saucer appear on the top line. If you want it to be further down the screen, you can precede the first line of printing with PRINT AT 5,0: " "; to shift the print cursor before the first line of the pattern is printed, or alternatively, you can use PRINT AT in place of PRINT TAB for the first line of the pattern. This method is one of two main methods which create shapes directly from the keyboard. If you now want to animate your graphics pattern, you will also have to create a blank which will fit over your graphics pattern to delete it, and then program a loop which rubs out and then creates the pattern alternately along or up/down the screen depending on the direction of motion that you want. This technique is illustrated later.

An alternative method of printing graphics, which is very much more useful for animation, consists of putting the whole pattern into a single string, just as we would do for a string of letters. On some computers, a string of this type can be made using the cursor-movement codes (8, 9, 10, 11), but this method is not applicable to Spectrum, because only code 8 works when used in PRINT CHR\$ n commands. As usual, Spectrum provides a more useful alternative, which is illustrated in Fig. 8.3. The graphics for another flying saucer shape are assembled

```
10 LET g$=" "
20 FOR y=1 TO 18: PRINT AT y,1
   " " : PRINT AT y+1,13;g$:
NEXT y
30 GO TO 30
```

Fig. 8.3. Creating a string of graphics characters rather than PRINTing directly.

into one string, g\$, which is typed using the computer in graphics mode. This is done in the usual way, typing as far as the quotes in the normal way, and then entering graphics mode (G cursor showing) for the characters. The shape is typed by using characters and blanks until complete – you can keep typing characters into the two bottom lines until the computer refuses to take any more. This won't happen until you have typed 23 lines, and no shape is likely to need so much typing – this is a facility which no other family of computers can offer. In the example, the steps needed to animate the shape are also shown alternately printing a wide blank and then the shape so that the top of the saucer is wiped each time the shape moves down. Animations of this type are simple in BASIC, and if they are accompanied by sound effects (see Chapter 12) they can be very effective. Note that in this example, the computer has been forced into an endless loop in line 30. This suppresses the message which is printed to inform you that the program has ended, and it is a useful tip for graphics programs, where the message spoils the impact of the graphics.

Creating a string from the keyboard in this way is by far the simplest way of making up a pattern from the LRG characters, but sometimes you may not wish to show to anyone reading the listing what is being created. The other method that

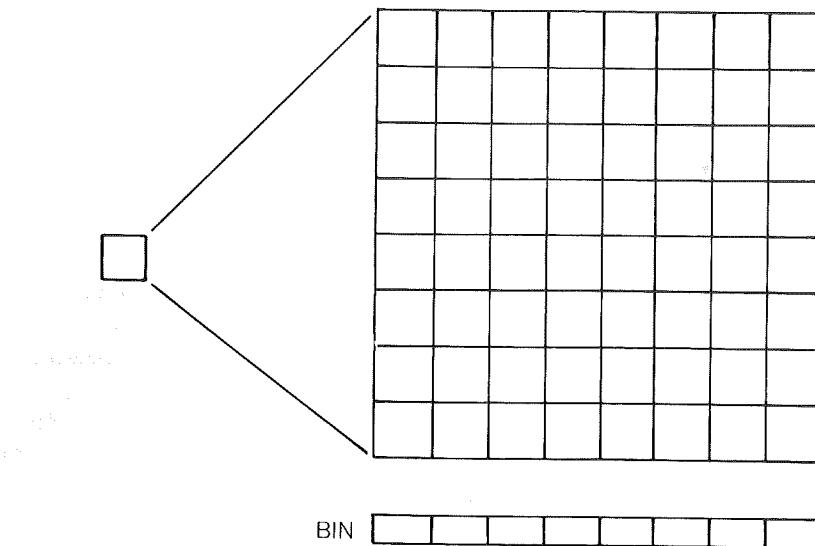


Fig. 8.4. An 8×8 pattern for creating your own shapes – use tracing paper over this.

can be used is to pack up a string using `CHR$` numbers. Packing up, in this context, means starting with an empty string and adding `CHR$` numbers until the string contains enough to form a pattern when printed. There is a `CHR$` number for each character, as indicated in Fig. 8.1 and in the manual, Appendix A. The important characters can be put in using `READ ... DATA`, and repeated characters, particularly blanks, can be put in using a `FOR ... NEXT` loop so as not to waste too much space in the `DATA` line. An (imaginary) example might read:

```
LET g$ = " "; READ a$: LET g$ = g$ + a$: FOR n = 1 TO 27:  
LET g$ = g$ + CHR$ 128: NEXT n: FOR n = 1 TO 7: READ a$:  
LET g$ = g$ + a$: NEXT n
```

In general, this is a lot of work and entry from the keyboard is preferable.

User-defined graphics

User-defined graphics means creating your own graphics characters, and this allows you to create much more interesting patterns with much higher resolution than can be obtained using the pre-defined characters that are on the keyboard. The price that you pay for this facility is, as you might expect, much more programming. Twenty-one different shapes of your own can be created and used in programs, saved and loaded, and used in every way like normal characters, by using the graphics mode along with the letter keys `a` to `u` inclusive. These characters can be printed either by using `PRINT "` and then slipping to graphics mode to enter the letter, then back to normal for the final quote mark, or by using their `CHR$` numbers, which are 144 to 164 inclusive. The character shapes are mapped onto a set of 8 by 8 squares, such as is shown in Fig. 8.4 – this can be used

as a template for designing shapes, using tracing paper over the squares.

The first thing to do is to decide what shape you want to use with some specified letter key (`a` to `u`). Let's suppose that we want the '`a`' key to give us a humanoid-cum-space-invader type of shape, as shown in Fig. 8.5. The first thing to do is to draw the shape, as shown, on the pattern of 8×8 squares. You will have to decide

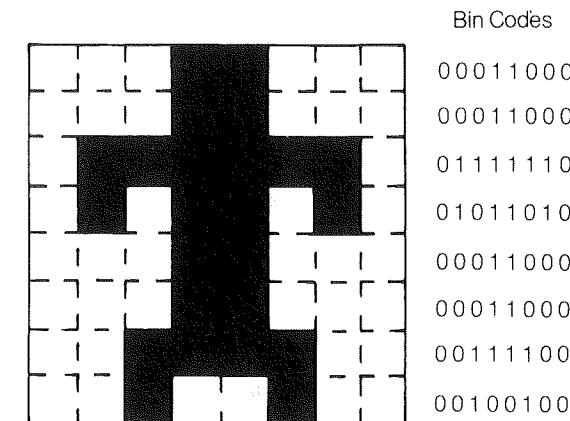


Fig. 8.5. Planning out a humanoid shape (or 'draw, Marshal...').

for yourself how you want to do this. If you draw to the edges of the 8×8 square outline, your character will merge with anything in the next line or space – this may be useful. If you leave a set of squares along the edges, leaving you working with a 6×6 set, then your character will remain clear of any others printed next to it. For many purposes in graphics programs, you do not need a margin, and the greater freedom that you have by using 8×8 (64 squares) rather than 6×6 (36 squares) is more useful.

Figure 8.5, then, shows our pattern drawn on to the set of squares. Remember that this is shown on a large scale – the whole pattern will only be the size of one character square when it is put on the screen. The pattern now needs to be converted into a set of number codes, and the simplest method is to use 'binary codes', where `0` means white (paper colour) and `1` represents black (ink colour). Numbers which use just these two digits are called *binary numbers*, and since these are the form of numbers that all computers use anyhow, it would be a useless type of computer that did not provide for entering numbers in this form. Needless to say, this is something that Spectrum can do, using an instruction `BIN` (for `BI`nary) to indicate that what follows is in binary code of `1`'s and `0`'s.

Going back to Fig. 8.5, if we look along the top line of our 8×8 set of squares, starting at the left-hand side, we see that the first three squares are blank (`0`), the next two are filled in (`1`) and the last three are empty (`0`). We write the coding for this as `00011000` in binary, and the same pattern is used for the second line. The other lines are treated in the same way, giving the set of `BIN` codes that are shown in the drawing.

So now we have a description of the pattern in language that the computer can understand, a set of eight BIN numbers. The next step is to instruct the computer so that one of the programmable keys can be used to produce this pattern. This is done by a set of instructions which place the 0's and 1's into the correct places in the memory of the Spectrum, and they involve using the new instructions, POKE and USR. First of all, we select a key that we want to use for this pattern – the 'a' key seems as good as any. USR is an instruction which is used to find places in the memory of the Spectrum, so USR "a" finds the first of a set of eight pieces of memory, each of which can hold eight 1's or 0's, and which will all be used to store the pattern for the 'a' key in graphics mode. This leaves the normal action of the 'a' key in the other modes unaffected.

If we place the binary number for the top row of our pattern in the first memory space found by USR, the second binary number in the next memory space, and so on, then this completes the programming of a key and from then on, until the computer is switched off or the same key re-programmed, the graphics pattern will appear when we enter graphics mode and press the key – if we press the key in graphics mode before the programming is carried out we will get the upper case letter, A in this example. Typing NEW (ENTER) has no effect on the programming of the key, though it will clear out the program that produced it!

Since numbers like BIN00011000 are used in groups of eight to program a key, the most convenient way of programming is to make use of READ ... DATA. USR is then used to find the correct starting-place in the memory, and POKE is used to place the BIN numbers into the memory. A piece of program such as:

```
FOR n = 0 TO 7: READ k: POKE USR "a" + n, k: NEXT N
```

will do nicely, with the DATA line or lines containing the BIN numbers. It's easier to see what is happening or to change lines if you use one DATA line for each BIN number. Using this method also allows you to use different sets of data lines, using RESTORE so that you can re-program your key at different stages in a program. Another hint, illustrated in the manual, is to allocate a variable name to a BIN number that you use several times, such as LET z = BIN00000000 or LET b = BIN 11111111 and then use POKE USR "a" + 5, z or POKE USR "a" + 6, b to place these lines into your pattern.

What the instruction does, then, is to find a place in memory, from the USR "a" part, and to place a number there using POKE. The loop can be used because USR "a" + 1 is the next memory place along from USR "a", so that the next line can be poked there. The loop uses n = 0 TO 7 rather than 1 TO 8 because you want the first binary number in USR "a" rather than in USR "a" + 1. This newly defined character can then be printed either by using its CHR\$ number (see the manual, Appendix A) or by pressing its key, 'a' in this example when the computer is in graphics mode and the G-cursor is showing. This complete procedure is shown in Fig. 8.6, but note that when you enter line 30 at first, it will read PRINT "A", because the shape is not created until the program has run – the illustration was printed after the program had been tried, like all of the others in this book.

To save your character for use the next time you want it, however, is not like

```
10 FOR n=0 TO 7: READ k: POKE
USR "a"+n, k: NEXT n
30 PRINT "A"
100 DATA BIN 00011000,BIN 000111
000,BIN 0111110,BIN 01011010,BIN
N 00011000,BIN 00011000,BIN 0011
100,BIN 00100100
```

Fig. 8.6. A program for the humanoid/invader/cowboy.

saving variables. If you have a line: LET a\$ = "A", where the A is in *graphics* mode, so that PRINT a\$ will produce the character, then you can save the whole program and expect that the value of a\$ will be saved. It will, but NOT as your special programming. When you load back and type PRINT "A", going into graphics mode for typing A, then all you get is A, not your shape. If you want to have user-defined shapes in a program, then the program must create the shapes unless you deliberately use a different type of load and save command of this form:

```
SAVE "blob" CODE USR "a", 8
```

where "blob" has been chosen as the 'filename' for this shape created by the program of Fig. 8.6, and CODE is a way of indicating to the Spectrum that what is to be stored is not a BASIC program but just a piece of code in the memory. USR indicates, as usual, where the start of this piece of memory is to be found (its *address*), and 8 is the number of bytes (sets of eight binary digits), of memory, since we have used eight sets of digits for each character. The corresponding VERIFY command for this type of SAVE is: VERIFY "blob" CODE and if you are loading the character back into your own Spectrum, then you can simply use: LOAD "blob" CODE. If you want to load your character into another Spectrum, however, particularly if it's a 48K one, it's safer to use: LOAD "blob" CODE USR "a" in case the other machine uses its memory differently.

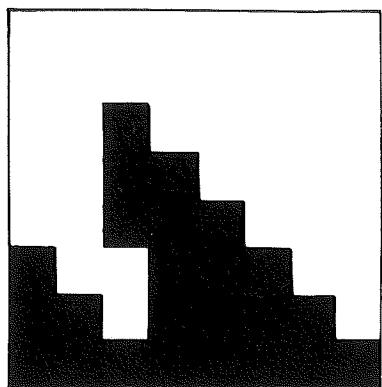
One thing which inevitably causes some surprise when user-defined graphics are created is how small each defined character is. This is ideal for many purposes, but sometimes you want to use larger shapes. This can be done by combining two or more defined characters and printing them together so that they appear as one. If each one is defined right up to one edge, then the edges can be made to merge when the characters are printed together (which can be done by defining them as one string as we used earlier with the LRG characters). Take a look at the two patterns in Fig. 8.7 and the BIN codes which produce them. This pair gives a reasonable fighter-plane appearance, and if the two characters are put into one string, then the plane can be moved across the screen in a fairly convincing way – the methods have already been illustrated. There is no reason why several sets of user-defined graphics shapes cannot be combined in this way, so that some very interesting shapes can be put on to the screen.

Plot and draw

Another way of drawing larger figures is to make use of the two main HRG commands PLOT and DRAW. PLOT means illuminate one point from all the possible 'squares' on the screen, of which we have 8 × 8 in each character position.

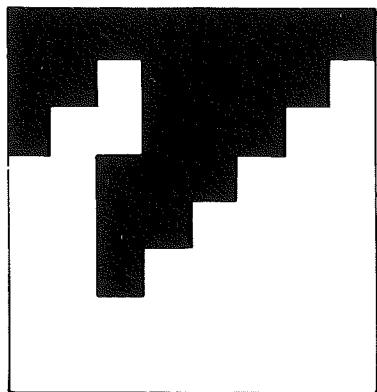
When you use PLOT or DRAW you are working with a screen which consists of 256 points across (numbered 0 to 255) and 176 vertically (numbered 0 to 175). A complete screen therefore uses 256×176 points, a total of 45056 points – and that's what we mean by high resolution. Any line that we draw will only be as thick as each point.

PLOT lights up the point whose across (x) and up (y) numbers follow the PLOT



BIN Codes

| |
|----------|
| 00000000 |
| 00000000 |
| 00100000 |
| 00110000 |
| 00111000 |
| 10011100 |
| 11011110 |
| 11111111 |



BIN Codes

| |
|----------|
| 11111111 |
| 11011110 |
| 10011100 |
| 00111000 |
| 00110000 |
| 00100000 |
| 00000000 |
| 00000000 |

Fig. 8.7. Creating two graphics shapes which can be printed together.

instruction, with the numbers separated by means of a comma. You must be careful to avoid confusing PLOT numbers with PRINT AT numbers. The PLOT numbers start at the *bottom* left-hand corner of the screen, which has the PLOT numbers 0, 0, and they read across (left to right) and upwards. The PRINT AT numbers start at the *top* left-hand corner, and also read across and *down*. PLOT uses numbers up to 255 across and 175 up, whereas PRINT AT uses up to 31 across and 21 down. Most important, PLOT uses its numbers in across, up (x,y) order; PRINT AT uses them in down, across (y, x) order. The differences are

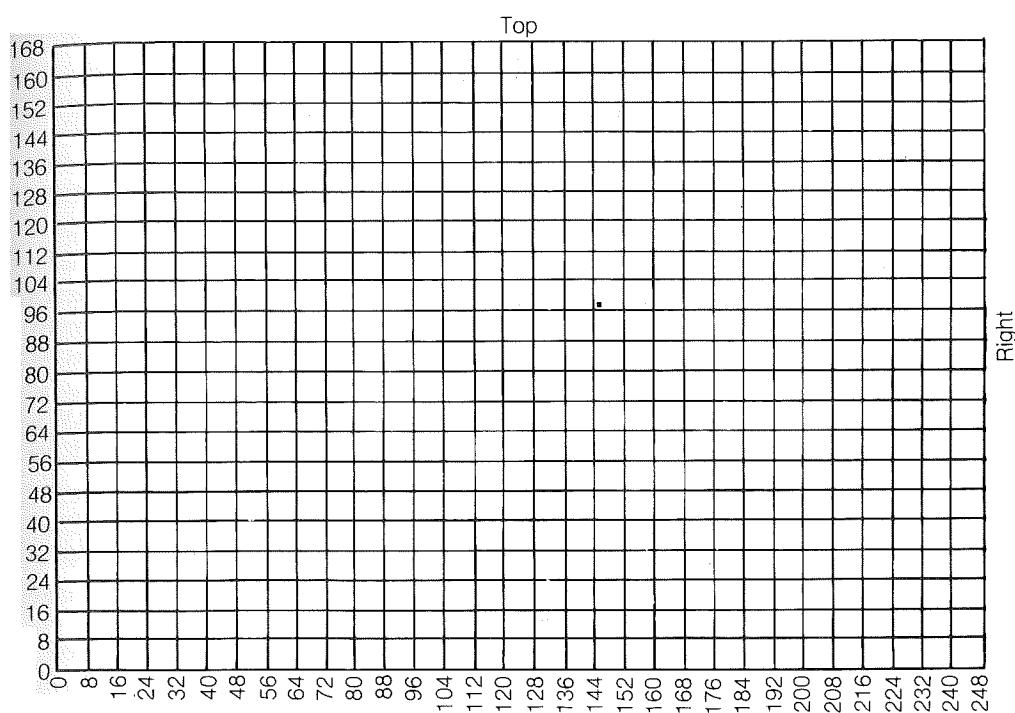


Fig. 8.8. The mapping of the screen for PLOT and DRAW instructions.

important, because text and drawings can be mixed on the screen of the Spectrum. Figure 8.8 shows the 'mapping' of the screen for the PLOT instruction.

PLOT 0, 0 will therefore illuminate a tiny portion on the bottom left-hand corner of the screen area (not on the lines reserved for commands). We can PLOT anywhere on the usable screen, but using an impossible number, as for example in PLOT 5,200 or PLOT 300,100 will result in the error message 'Integer out of range'. We can use PLOT to go to the starting point of a drawing, or print one point at a time, as is needed for graphs.

DRAW means draw a line from your starting point to some new point. Your starting point will be where a PLOT instruction placed a point, or where a previous DRAW instruction left off. Positive numbers used with DRAW will draw a line to the right and up from the PLOT point, negative values will draw a line which moves left and down. Note that negative numbers cannot be used with PLOT, only with DRAW. Once again, if the number that is used would cause the line to be drawn off the screen, the 'Integer out of range' message will come up.

Take a look now at some examples. Figure 8.9 shows examples of lines drawn vertically, horizontally and diagonally, simply to show how the commands work. It's very tempting to treat DRAW as if it were like PLOT – but it's a temptation that you have to resist! When you design HRG drawings, you will find it easier if you note the co-ordinates (x and y numbers) of each end-of-line position, and remember that the numbers you will need for the DRAW instruction are obtained by subtraction. For example, if you ended a line at position 240,150 (240 across

```

10 PLOT 0,0
20 DRAW 255,175
30 GO SUB 500
40 DRAW 0,-175
50 GO SUB 500
60 DRAW -255,0
70 GO SUB 500
80 DRAW 0,175
90 GO SUB 500
100 DRAW 255,0
110 GO TO 9999
500 INPUT "Press ENTER to proce
ed.":x$: 
510 RETURN

```

Fig. 8.9. Using PLOT and DRAW.

and 150 up), and you want to go to 100,160, then the command is DRAW -140,10, which is new x - old x, new y - old y. Life would have been a lot easier if we could have used absolute co-ordinates!

Using DRAW to produce straight lines is rather limiting, so an extension to the command allows a third number to be added so as to permit DRAW to produce a circle, or any part of a circle. The first two numbers specify the finishing point of the line as before, but the third one specifies angle in radians (see Fig. 8.10). The radius of this circular portion depends on the distance between the starting point and the finishing point *and the angle*, so it's never obvious, until you have had a lot of experience with the command, where the circular part will pass. An example is shown in Fig. 8.11, which draws two semicircles to form a horizontal S – note that the direction of drawing the circular portion is determined by the sign (+ or -) of the number used to specify the angle.

To make the best use of these commands in HRG, place a piece of tracing paper

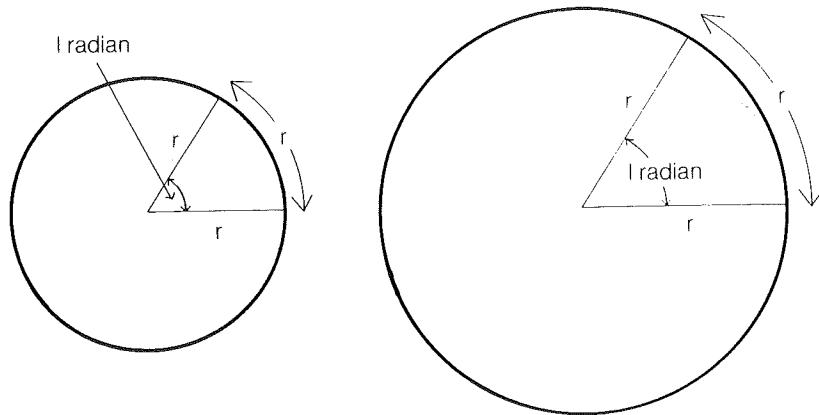


Fig. 8.10. Radian measures for angles. A radian is a natural unit of angle based on the circle. For a circle of any size, if you take a piece of the circumference whose length equals the radius, r , then drawing a radius from each end of this piece produces an angle which is equal to 1 radian. PI radians equal 180°, so that 1 radian = 57.29° and 1° = 0.01745 radians.

```

10 PLOT 240,170
20 DRAW -120,-80,-PI
30 DRAW -120,0,PI

```

Fig. 8.11. Drawing portions of circles – note the effect of sign.

over a 256×176 grid, such as the one in Fig. 8.8. Draw your shape or shapes, remembering that the most effective drawings are usually those which avoid too much fine detail. Having sketched out the drawing, break it down into a series of straight lines and circular portions, and mark co-ordinates (x across and y up) for each starting/finishing point. Select a starting place for POINT (more than one, if there are several unconnected shapes), find its co-ordinates, and then start working out your DRAW numbers. You will probably want to modify your numbers when you see the drawing on the screen, because it always looks rather different, but if your planning was reasonable, the changes should not be extensive. Where a full circle is needed, there is a CIRCLE command, which requires three numbers, the x and y co-ordinates of the centre and a number for the radius. Figure 8.12 shows an example of the use of this command to draw a 'spider-web' pattern – it also shows the deficiencies of the CIRCLE command, because some of the circles are not very circular!

```

10 PLOT 125,88
20 FOR n=1 TO 68 STEP 2
30 CIRCLE 125,88,n
40 NEXT n

```

Fig. 8.12. The CIRCLE instruction – not perfect, but useful.

Screen drawings, which can take a considerable amount of programming, can be saved on tape, so that they can be loaded separately, to avoid having to use up a lot of program memory to create the drawing again. The save command takes the form:

SAVE "graphics" SCREEN\$

and this takes quite a long time to save, considerably longer than to SAVE the program which creates it. To load this back, you will need to use:

LOAD "graphics" SCREEN\$

and this also will take quite a long time, but is fascinating to watch because it illustrates how the screen images are stored in the memory (see the introduction to this section). You *can't* use VERIFY on any of these drawing-saving routines, because the screen appearance usually changes between saving and verifying, resulting in a false error message.

PLOT can be used apart from DRAW to plot out shapes which can be expressed in mathematical equations. Any equation which uses variables x and y and which can be 'fiddled' so that the values of x and y do not exceed the limits of 0 to 255 for x and 0 to 175 for y can be used. Figure 8.13 shows one example of a program which draws a trace in this way, and also illustrates how slow this type of PLOT graphics can be. Note the use of $(\text{SIN } x)^*(\text{SIN } x)$, because the Spectrum

```

10 CLS : FOR x=0 TO 254
20 LET y=100*(SIN x)*(SIN x)
30 PLOT x,y
40 NEXT x

```

Fig. 8.13. A graphical plot – it takes a long time!

will not accept the instruction $(\text{SIN } x)^2$ when the value of the SIN is negative – this is an oddity of Spectrum which is not found on other computers. You can, however, experiment with how changes of numbers and of STEP values in the loop will change the appearance of the shape.

```

10 PRINT AT 10,5;"Normal"
20 INVERSE 1
30 PRINT AT 10,5;"Inverse"
40 INVERSE 0
50 PRINT AT 15,5;"Restored"

```

Fig. 8.14. Using INVERSE in separate lines – the effect persists until reversed.

INVERSE, OVER, FLASH and BRIGHT

INVERSE 1 has the effect of inverting the dot pattern of any character. When you have the situation, as exists at switch on, of black characters on a white background, INVERSE 1 will cause the pattern to be inverted to white dots on a black background. This does not mean that the whole screen will turn black, unless you have written in every part of it, only that each character will be made up from white dots on a black background for that character. INVERSE must always be followed by the digits 1 or 0 – 1 means do the inversion, 0 means cancel, and the command can be placed in a line, as illustrated in Fig. 8.14, or after a PRINT instruction, as shown in Fig. 8.15. When INVERSE 1, or any of the other instructions of this group, OVER, FLASH, BRIGHT, is included after PRINT, its effect lasts only for the duration of that PRINT instruction – the next colon or the next line number will restore normality.

```

10 PRINT "Normal"
20 PRINT INVERSE 1;"Inverse"
30 PRINT "Normal again"
40 PRINT INVERSE 1;"Inverse";
INVERSE 0;"Normal"

```

```

Normal
Inverse
Normal again
InverseNormal

```

Fig. 8.15. Using INVERSE 1 in a PRINT statement – the effect lasts only within the statement.

Looking now at FLASH and BRIGHT, these cause, respectively, flashing characters or extra-bright characters, and can be used inside or outside PRINT

statements as illustrated in Fig. 8.16. Like INVERSE, when BRIGHT or FLASH is used after PRINT, then its effect lasts only for as long as the PRINT statement is in operation.

OVER 1 is a particularly interesting instruction which is not found on many computers. The normal PRINT action will delete any character in a printing position before a new character is printed there. Adding OVER 1 to a PRINT

```

10 CLS : PRINT TAB 9;"Flash &
Bright"
20 PRINT "Normal line of text
"
30 BRIGHT 1
40 PRINT "this is bright"
50 PRINT "so is this"
60 BRIGHT 0
70 PRINT "Normal again"
80 FLASH 1
90 PRINT "Flashing"
100 PRINT "as is this"
110 FLASH 0
120 PAUSE 50
130 PRINT BRIGHT 1;"This is bri
ght, but this"
140 PRINT FLASH 1;" flashes."
150 PRINT " and this doesn't"

```

Fig. 8.16. Using BRIGHT and FLASH. BRIGHT does not always show up well on TV receivers.

instruction will cause superposition of characters, a feature which has some use in creating new characters, adding accent marks, and underlining in printed text and can also be used extensively in graphics. Figure 8.17 shows an example of this instruction which can be countermanded by OVER 0 either inside or outside a PRINT statement. One use I find particularly handy is underlining text; but another is in animation, because a graphics shape can be changed gradually by overprinting another shape, and then changed by replacing with another shape, or the same shape printed along, up or down. Some practice is needed before you get too ambitious with this one, however.

```

10 CLS : PRINT TAB 13;"Over 1"
20 PRINT AT 5,5;"Leave"
30 PRINT AT 5,8; OVER 1;""
40 PRINT AT 7,15;""
50 PRINT AT 7,15; OVER 1;"-"
60 PRINT AT 10,11;"Underlined"
70 PRINT AT 10,11; OVER 1;""
80 PRINT AT 10,11; REM Underline on key Z
90 PRINT AT 10,11;""
90 PRINT AT 10,15; OVER 1;""

```

Fig. 8.17. Printing one character over another, using OVER 1.

Chapter Nine

Getting it in Colour

The eight colours of the Spectrum are printed, as a reminder, on the top line above the top row of keys – they include Black and White, but the count is genuine – a flashing colour is not counted as being different from a steady colour the way it is by some computers! On a black and white receiver, these colours will appear as shades of grey ranging in intensity from 7, which is white, down to \emptyset , which is black, and a good B&W receiver should display them so that you can tell the difference.

There are three instructions which are particularly associated with colour. These are BORDER, PAPER and INK (see Fig. 9.1), and each of them needs to be followed by a number. There is no need to memorise these numbers, because the colours are printed above the number keys, so that 1 is blue, 2 is red, 3 magenta, and so on. Only 8 and 9 among the number keys have no obvious colour more about them later.

Program lines or printing
'INK' colour.

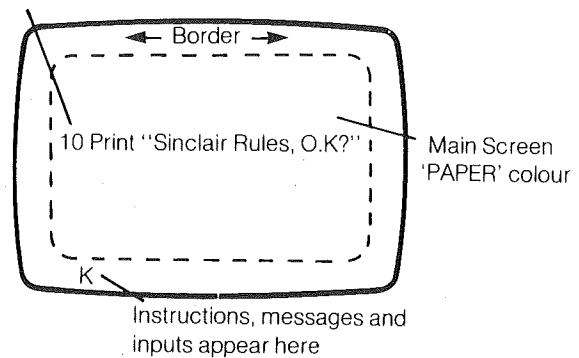


Fig. 9.1. How the screen is divided up into areas which can be of different colours.

The BORDER instruction affects the colour of the border around the usable part of the screen area – it is on the bottom section of this border that your lines appear as you type them in, and where you also type INPUT answers. Try the program of Fig. 9.2, which shows the range of BORDER commands, noting that the INPUT printing *always* appears in a contrasting colour to the border without any programming effort on your part – this is deliberately built into Spectrum so that no matter what you do, you can always see the messages in the bottom portion of the screen.

```

10 CLS : PRINT TAB 9;"Floral B
orders"
20 FOR n=1 TO 7
30 GO SUB 300
40 BORDER n: NEXT n
50 PRINT "That's all!": PRIM
T AT 12,16;7: GO TO 9999
60 PRINT AT 12,16;n-1: INPUT (
"For border colour");n): PRESS E
NTER,":,$$: RETURN

```

Fig. 9.2. The range of BORDER colours

BORDER is mainly used as a 'picture-frame' instruction, and the two colour instructions which affect the main screen area are PAPER and INK. As you might expect from the names, PAPER affects the background colour and INK affects the colour of the printing on that background; the appearance can be inverted by using INVERSE 1 as before. These effects can be used directly, but if you type PAPER 1 to get blue paper, you will need to press ENTER *twice* before you see any effect. This is because the command consists of two sections, one for PAPER, the other for the number. No special treatment is needed, however, when the instructions are used within a program, as Fig. 9.3 illustrates. This prints a

```
10 CLS : BORDER 1
20 FOR n=1 TO 7: GO SUB 300
30 PAPER n: PRINT INK 9;"#####"
: REM Graphics 6
40 PRINT ":" NEXT n: LET n=0: G
0 SUB 300
50 PRINT "";"That's all!": GO T
0 9999
300 INPUT ("For paper colour "):
0;" Press ENTER."): TAB: RETURN
```

Fig. 9.3. PAPER and INK used with graphics

graphics block in INK colour on the selected PAPER colour, and also illustrates how 'colour' 9 is used. INK 9 means *use a contrasting colour* and it ensures that the ink will always contrast with the paper. If you substitute text for the graphics block, then your text will also appear in colour, but on the early Spectrum models, text was not very clear when it was printed in this way because of a flickering effect. Some of this can be reduced by careful tuning of the colour receiver (strictly speaking, it needs some adjustment each time it is used, because the modulator of Spectrum does not send out a constant frequency signal); this type of pattern is ideal for checking that the tuning of a colour receiver is at its best. My own preference is to leave text of normal size as black on white, and to keep colour effects for graphics, in which role they are particularly effective.

An instruction which makes use of PAPER or INK affects a complete character position, made up of 8 dots across and 8 rows down (as you know from user-defined characters). In one character position, however, you can have only one PAPER colour and one INK colour, and that's your lot. For displaying blocks of colour, you can achieve some mixing by inverting adjacent blocks, and this technique is

illustrated in Fig. 9.4, using the low-resolution graphics to print a screenful which at a distance will appear to be orange. A more elaborate method is to have each element in an alternate colour (one PAPER, and the other INK) so as to produce

```
16 CLS : BORDER 1
17 FOR n = 1 TO 552
18 PRINT INK 6: PAPER 0, " "
19 PRINT INK 2: PAPER 6, " "
20 NEXT n
```

Fig. 9.4. Creating an impression of colour mixing

colours which are quite acceptable mixtures even on closer inspection – this is illustrated on page 124 of the Spectrum manual.

LRG patterns in colour

The use of colour in graphics adds a new dimension to the design of patterns, and one which can be varied cunningly by the use of INVERSE, BRIGHT and FLASH. Figure 9.5 illustrates how a checker-board pattern can be drawn in blue

```
10 LET 9$=" " : " " : " "
20 FOR J=1 TO 4: FOR X=1 TO 9
30 PRINT PAPER 1: INK 6;9$: NE
40 XT X: FOR X=1 TO 9: INK 1;9$: NE
50 PRINT PAPER 6: INK 1;9$: NE
60 XT X
70 NEXT J
```

Fig. 9.5. A checker-board pattern program.

and yellow by making use of a graphics string which is printed twice with one PAPER and INK setting, then twice with the settings inverted. The amount of programming can be reduced further by defining a string consisting of two paper and two ink squares, and using a FOR ... NEXT loop to produce each pair of lines. This technique is illustrated in Fig. 9.6.

```
10 LET S$="": BORDER 4
20 FOR I=1 TO 6: FOR X=1 TO 16
30 PRINT PAPER 1; INK 6;S$;: N
EXT X
40 FOR X=1 TO 16: PRINT PRPER
5; INK 1;S$;: NEXT X
50 NEXT I
```

Fig. 9.6. A version using a shorter string and a longer loop.

A very effective use of the colour LRG patterns is the production of random patterns on the screen, followed by a giant title made up from black graphics blocks. This is usable only for short title words, but it ensures the maximum attention for the title. A program of this type is illustrated in Fig. 9.7. The PRINT loop is set to cover the whole screen, with 32×22 lines = 704 positions generating a pattern in which the paper colour is chosen by random selection from the eight

```

10 CLS : FOR n=1 TO 794: LET j
=1+INT (RND*7): LET k=INT (RND*1
6)+128
20 PRINT PAPER j; INK 9; CHR$ k
: NEXT n
30 PAUSE 500: INK 0
35 PRINT AT 6,9;""
40 PRINT AT 9,9;" "
50 PRINT AT 10,10;" "
60 PRINT AT 11,10;" "
65 PRINT AT 12,9;" "

```

Fig. 9.7. Random colours used to bring attention to a giant title.

colours, using variable *j*, with INK taken as 9 to ensure contrast. The characters that are printed are the LRG characters, whose code numbers range from 128 to 143. Since there are sixteen such characters, we generate a choice of 16 random numbers, and then add this to 128 to get the sequence of character codes that we use for variable *k*.

Once this mottled piece of colour has been drawn, there is a ten second pause for you to admire it, or to phone the Hayward Gallery about it, and then the word TITLE is printed in large black letters at the centre of the screen. The word is generated from LRG characters printed with the TAB settings that keep each line of characters in the correct positions, with its own borders to wipe out the coloured pattern near the letters. Titles like this are generated in the same way as we use for any other graphics pattern – by drawing the letters out on graph paper which has clear 1 cm squares, and then shading in to show the graphics shapes which are then entered from the keyboard in this example.

HRG in colour

The PLOT and DRAW instructions can be used with your choice of PAPER and INK colours so that coloured lines can be drawn on coloured paper as easily as black on white or white on black. Colour is not so detectable in small points, however, and the effect is not so useful as it would be if it were possible to fill in a shape with colour by a simple command and see the outline clearly. When a paper or ink colour is fixed, however, it affects all of the points in a block, so that where two coloured lines come within the same character block, the PAPER and INK colours in that block are affected by the last set of PAPER and INK instructions, so that the lines may appear to 'blot'. This is particularly noticeable if several lines intersect close to each other, making the lines appear thick where they are close to each other.

POINT and ATTR

POINT and ATTR are not so much graphics instructions as 'discovering'

instructions. They provide information about character blocks, but in rather different ways, and they are of more use in maze, wall and bat'n'ball games programs – though BASIC is too slow for bat'n'ball generally.

POINT has to be followed by two co-ordinate numbers within brackets, in x,y order (across, up), separated by comma and using the normal HRG numbers of 0 to 255 across and 0 to 175 up. POINT causes a value to be returned, and that value, which can be printed, or tested using IF ..., will be 0 if the block is at paper colour and 1 if it is at INK colour. In this way, it is possible to detect a boundary line by using a statement such as:

IF POINT (x,y) = 1 THEN ...

An example of POINT used in this way is illustrated in Fig. 9.8. A point which is lit by using PLOT is moved horizontally by using LET x = x + k. Since k = 1 at

```

10 PAPER 1: INK 6
20 CLS: CIRCLE 128,66,50
30 PLOT 128,66: LET x=0: LET k
=1
40 PLOT INVERSE 1,128+x,66: LE
T x=x+k
50 PLOT 128+x,66: IF POINT (12
8+x+k,66)=1 THEN LET k=-1*k
60 GO TO 40

```

Fig. 9.8. Using POINT to detect boundaries.

the start of the program, this movement will be from left to right, and the point which has just been vacated is reset to paper colour, so as to give the impression of a crawling dot. Each new position is tested, using POINT to see if it corresponds to the boundary of the circle. If it does, then the direction of motion is reversed by using LET k = -k, so that the x co-ordinate number is reduced on each pass through the loop. Once again, each point is tested to check for the boundary line, and when this is found, LET k = -k again reverses the sign of k (two negatives multiplied give a positive), and the action repeats. It's relatively simple, but it demonstrates how POINT can be used, and that's what this section is about.

One item may be of interest to potential game-makers. Suppose you have vertical walls drawn on the screen, and an object approaching with both x and y values varying. To make a lifelike bounce (neglecting gravity) you should reverse the x values (go from x = x + 1 to x = x - 1) when the wall is struck, but leave the y values to change as before. It works in the opposite way when an object strikes a horizontal wall – you then reverse the changes of y values, and leave the x values to change as they did before.

The ATTR instruction is used to find out everything that is going on in a *character block* rather than at a single point in the block. Unlike POINT, which tests an individual point to see if it is PAPER or INK colour, ATTR tests to see what actual PAPER and INK colours are being used, whether extra brightness is set, and if the character is flashing. ATTR must be followed by the same type of line and column numbers, within brackets and with a separating comma, as

PRINT AT, with the range of θ to 31 columns and θ to 21 for lines – remember that the numbers and the arrangement are different to those used for POINT.

Using ATTR results in a number, and you then have to analyse this number to find what is going on within the character block. The analysis goes as follows:

If ATTR (y,x) is equal to 128 or more, then the character is flashing, otherwise it is not.

If the ATTR number is greater than 127, then divide it by 128 and look now at the remainder – or if it was less than 127 in the first place, look again at the number.

Next question – is it greater than or equal to 64? If it is, then the character is BRIGHT, so note that, divide by 64 and use the remainder. If it is less than 64, carry on anyhow.

Divide what's left by eight. The whole number quotient (what you get by dividing) is the PAPER colour, and the remainder is the INK colour. Figure 9.9 shows a typical analysis, and Fig. 9.10 shows a program written to carry out this analysis, so saving you from all this work. You enter the ATTR number into the program at the INPUT step, and you will get a list of the attributes on the screen.

ATTR number 179

This exceeds 128, so block is **flashing**.

$$179 - 128 = 51$$

This is less than 64, so block is **not** bright.

$$51 \div 8 = 6 \text{ and } 3 \text{ remains.}$$

So paper colour is 6
ink colour is 3

Fig. 9.9. Analysing an ATTR number.

```
10 CLS : PRINT TAB 11;"Attributes": PRINT "Please enter attribute number"
20 INPUT n: LET n=INT n
30 IF n>127 THEN PRINT "Flashing"
40 LET n=n-128
50 IF n>63 THEN PRINT "Bright"
: LET n=n-64
50 PRINT "Paper colour is ";INT (n/8)
60 PRINT "Ink colour is ";8*(n/8-INT (n/8))
```

Fig. 9.10. A program for analysing an ATTR number.

Chapter Ten

Drawing Techniques

Now that you have had a chance to familiarise yourself with the new commands you need to make the most of Spectrum graphics, you're ready to try some more advanced ideas. This chapter should help you to expand your repertoire of graphics routines: it won't show you all there is to know, but it will give you a series of useful program ideas that can be applied in many different situations. It will also make use of some of the mathematical and trigonometric functions available from your keyboard – these keys are very much more useful than they sometimes appear to be!

Although the Spectrum's own CIRCLE command can draw circles for you till the cows come home (or the power unit blows up), it's still extremely useful to learn other ways of drawing circles, mainly because the methods used can then be adapted to drawing other, more interesting shapes. If you have forgotten all the trigonometry you ever knew, don't worry – this chapter is a self-contained revision course!

Circles by trigonometry

This method traces out a circle by drawing a series of short lines. Figure 10.1 shows a segment of the circle – points B and C are on the circumference of the circle, and point A is at the centre, so the sides AB and AC are both equal, because both are radii. To approximate the segment of the circle we shall simply draw the line BC. If we take the co-ordinates of point A as cx, cy, then the co-ordinates of point B are cx + (length AB), cy. The co-ordinates of point C, however, are harder to calculate. Let us drop a vertical line from C to point D. The x value for point C is given by the length of line AD and the y value is equal to the length of line CD. This is where the trigonometry comes in.

We will call the angle at point A of the triangle *theta* (θ) which is the name of the Greek letter normally used for labelling angles. In our program we shall use the variable name 'th' to represent the angle theta.

To find the length of side CD the function we need to use is SIN(theta). The definition of SIN(theta) is that it is the ratio of the length of the side of the triangle opposite angle theta to the length of the *hypotenuse* (the side opposite the right angle) which is side AC. So in our triangle:

$$\text{SIN}(th) + CD/AC$$

We already know that AC = radius r. The length of side CD is the change we need

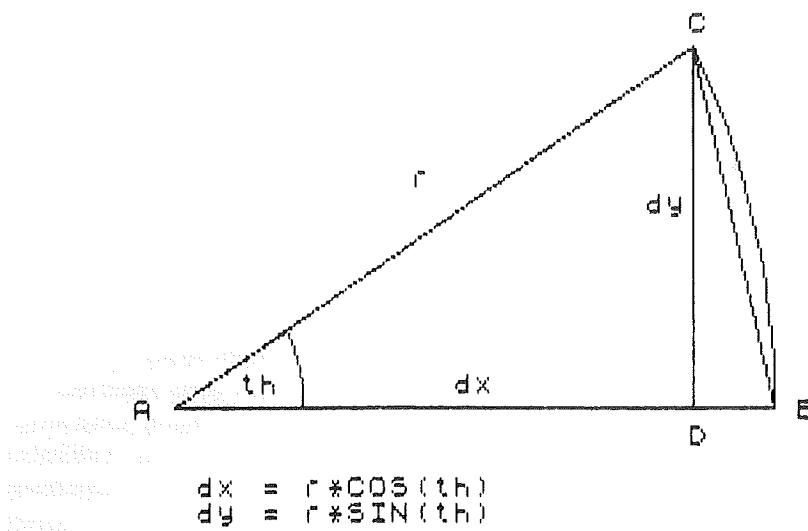


Fig. 10.1. Derivation of trigonometric method for circles.

to make to y to give the new y value for point C . We shall call this dy . Substituting these new terms in the equation we get:

$$\sin(\theta) = dy/r$$

and if we multiply both sides by r the result becomes:

$$dy = r * \sin(\theta)$$

Having found dy we need to find a value for side AD which is the required change in x and which we shall call dx . Now it just happens that $\cos(\theta)$ is the ratio of the length of the adjacent side (AD) of the triangle to the length of the hypotenuse (AC) so we get:

$$\cos(\theta) = AD/AC$$

and substituting the values dx and r gives:

$$dx = r * \cos(\theta)$$

To find the co-ordinates for the next point on the circle we apply the same equations but now angle θ has a different value.

To draw the circle we must first of all place the cursor at point B for which $dx=r$ and $dy=0$. This step is carried out by first setting variables $x1= cx+r$ and $y1= cy$ then using:

PLOT x1,y1

to plot the first point. The variables cx and cy are the co-ordinates for the centre of the circle.

The next step is to calculate the co-ordinates of point C which are given by the equations:

$$x2 = cx + dx = cx + r * \sin(\theta)$$

$$y2 = cy + dy = cy + r * \cos(\theta)$$

and using $x2,y2$ we can draw the line BC by using:

DRAW x2-x1,y2-y1

For the next line segment of the circle the values of $x1$ and $y1$ are set equal to the values of $x2$ and $y2$. The angle θ is then increased and new values are calculated for $x2,y2$ using the new value for the angle θ . This process continues until the angle θ reaches 360 degrees when a complete circle will have been drawn.

How do we decide on a value for θ ? Well there are 360 degrees in a complete rotation of the angle θ around the circle. To draw a smooth circle the more steps we use the better. A practical value for the number of steps is the number of units of radius r . Suppose we want a circle of radius 60. In this case, each segment of the circle adds $360/60$ or 6 degrees to the value of θ .

Whilst angles in degrees are familiar to us, the computer doesn't work in degrees but uses radians instead. In Chapter 8 we saw that 360 degrees is equal to 2π radians so therefore the angle increases by $2\pi/60$ radians for each segment of the circle.

The number π is a constant whose value is approximately 3.14 and it is the ratio of the circumference of a circle to its diameter. We do not need to remember the value for π because the Spectrum has a special key which allows us to insert π into a program statement as a constant. To get the term π into a statement, the CAPS SHIFT and SYMBOL SHIFT keys are pressed together to get extended keyboard mode and then the M key is pressed.

Drawing the circle involves using a simple loop to repeat the calculations and draw a short line segment r times. After each segment of the circle has been drawn the value of θ is increased and the values of $x1$ and $y1$ are updated to point to the end of the line that has just been drawn ready for the next drawing step.

To draw our circle we merely calculate a series of values of x and y for values of theta from 0 to 360 degrees (0 to 2π radians). The number of points we need depends upon the size of the circle and how accurately we want to draw it. A good figure to use is the number of units of the radius, so for a circle of radius 50 we might use 50 points. The angle theta for each step can be found by simply dividing 2π by the required number of points so the step size would be $2\pi/50$. A program for drawing circles using this technique is shown in Fig. 10.2. This program draws a series of random size circles all over the screen to give a result similar to that shown in Fig. 10.3.

The actual circle drawing section is written as a subroutine starting at line 500. If a number of circles is required it is convenient to use a subroutine for drawing the circle and call it from the main program whenever a circle is required to be drawn. Sometimes the calculated values for x and y co-ordinates may fall outside the limits of the screen and if used in a DRAW command would produce an error message and stop the program. To avoid this $x2$ and $y2$ are tested and if outside the screen limits they are set to the corresponding screen limit value. So if $x2 < 0$ then

```

100 REM Circle drawing using the
110 REM trigonometric method
120 BORDER 6
130 FOR s=1 TO 10
140 CLS
150 FOR n=1 TO 10
160 LET r=10+INT (RND*40)
170 LET x=INT (RND*255)
180 LET y=INT (RND*175)
190 GO SUB 500
200 NEXT n
210 PAUSE 100
220 NEXT s
230 STOP
500 REM Circle subroutine
510 LET dt=2*PI/r
520 REM Set starting point
530 LET th=0
540 LET x1=x+INT (r*COS th)
550 LET y1=y+INT (r*SIN th)
560 REM Correct off screen points
570 IF x1>255 THEN LET x1=255
580 IF x1<0 THEN LET x1=0
590 IF y1>175 THEN LET y1=175
600 IF y1<0 THEN LET y1=0
610 PLOT x1,y1
620 FOR i=0 TO r
630 LET th=th+dt
640 LET x2=x+INT (r*COS th)
650 LET y2=y+INT (r*SIN th)
655 REM Correct off screen points
660 IF x2>255 THEN LET x2=255
670 IF x2<0 THEN LET x2=0
680 IF y2>175 THEN LET y2=175
690 IF y2<0 THEN LET y2=0
700 DRAW x2-x1,y2-y1
710 LET x1=x2
720 LET y1=y2
730 NEXT i
740 RETURN

```

Fig. 10.2. Random circles using the trigonometric method.

x_2 is set to 0. This produces a line at the edge of the screen where part of the circle is outside the screen limit.

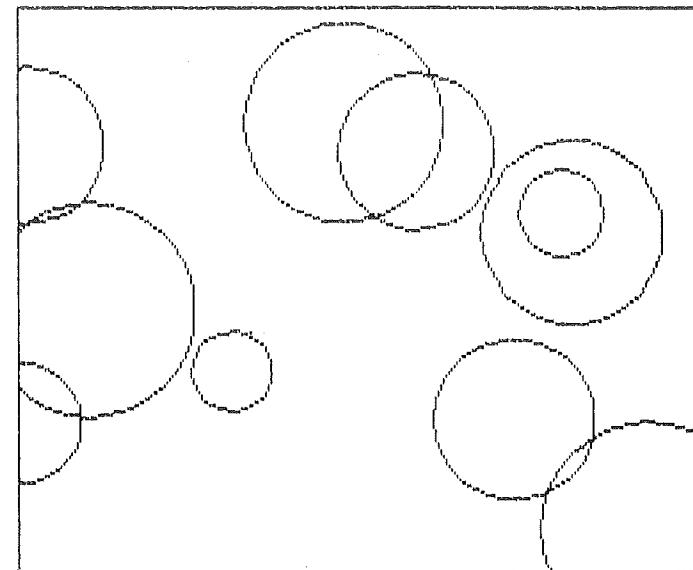
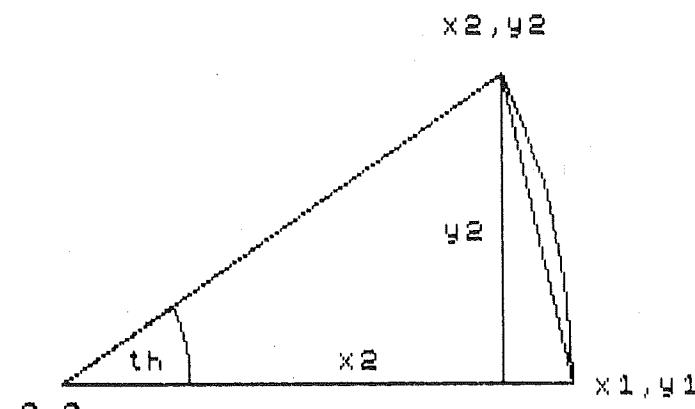


Fig. 10.3. Typical picture from program of Fig. 10.2.

The rotation method

A different approach to the calculation of the x, y values for a circle is to base them upon the angle through which the radial line is rotated at each step. In this case the new values for x_2 and y_2 are calculated from the values for the previous point (x_1, y_1) rather than from the radius and the total angle.



$$\begin{aligned}x_2 &= x_1 * \cos(\text{th}) \\y_2 &= x_1 * \sin(\text{th})\end{aligned}$$

Fig. 10.4. Rotation from the x axis.

If we look at Fig. 10.4 the value of y_1 is zero so that only the x_1 value, which also happens to be equal to r , affects the results. Here we get:

$$\begin{aligned}x_2 &= x_1 * \cos(\theta) \\y_2 &= x_1 * \sin(\theta)\end{aligned}$$

Now consider the situation where the radial line is vertical and is moved through angle theta. This is shown in Fig. 10.5. Here the value of x_1 is 0 and only the y_1

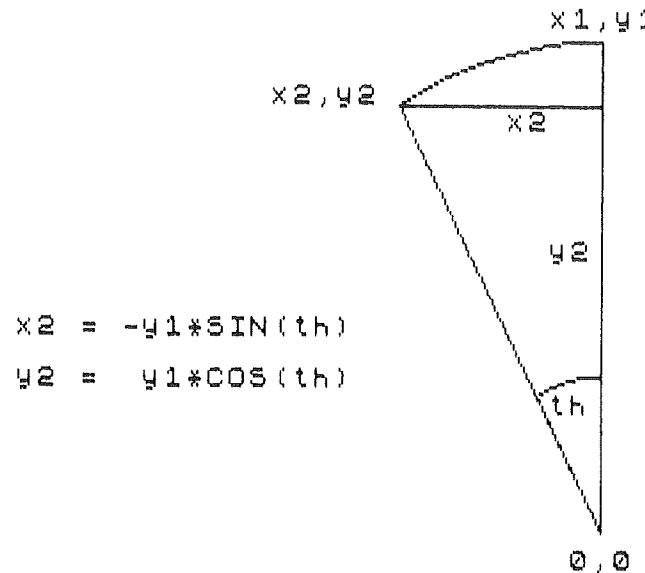


Fig. 10.5. Rotation from the y axis.

value affects the results. In this case the value of x_2 is negative since the point has been shifted to the left of the line where $x_1=0$. Here we get results:

$$\begin{aligned}x_2 &= -y_1 * \sin(\theta) \\y_2 &= y_1 * \cos(\theta)\end{aligned}$$

If we combine these two results we can produce a general expression for calculating x_2 and y_2 for any initial values of x_1 and y_1 . The two new equations are:

$$\begin{aligned}x_2 &= x_1 * \cos(\theta) - y_1 * \sin(\theta) \\y_2 &= x_1 * \sin(\theta) + y_1 * \cos(\theta)\end{aligned}$$

The big advantage of this approach is that the value of θ is constant so we can work out the values of $\sin(\theta)$ and $\cos(\theta)$ before entering the co-ordinate calculation and line drawing loop, thus eliminating virtually all the trigonometric calculations (which tend to be slow). The program for drawing a circle now becomes as shown in the listing of Fig. 10.6.

```

100 REM Random circles by the
110 REM rotation method
120 FOR n=1 TO 15
130 LET r=10+INT (RND*40)
140 LET x=r+INT (RND*(255-2*r))
150 LET y=r+INT (RND*(175-2*r))
160 GO SUB 500
170 NEXT n
180 STOP
500 REM Circle subroutine
510 LET th=2*PI/r
520 LET x1=r
530 LET y1=0
540 PLOT x+x1,y+y1
550 FOR i=1 TO r
560 LET x2=x1*COS th-y1*SIN th
570 LET y2=x1*SIN th+y1*COS th
580 DRAW x2-x1,y2-y1
590 LET x1=x2
600 LET y1=y2
610 NEXT i
620 RETURN

```

Fig. 10.6.. Circle drawing by the rotation method.

Drawing polygons

If we reduce the number of steps and hence the number of line segments used in the circle drawing routine the result will be a figure with a number of equal straight sides. Such a figure is called a regular *polygon*. If we use the trigonometric method the steps in the angle θ become quite large. Suppose we drew an eight-sided polygon, which is called an *octagon*, then the angle will change by $2\pi/8$ at each drawing step. The program listed in Fig. 10.7 will draw a single octagon at the centre of the screen. If we wanted a six-sided figure, known as a *hexagon*, then the number of steps in the drawing loop is reduced to 6. Thus the total angle of 2π is divided by 6 to give a change in θ of $2\pi/6$ for each step.

To make a general polygon drawing routine we could introduce a new parameter ns (number of sides) and then modify the program to make the required number of drawing steps. Thus the change in θ at each step (dt) will be given by:

$$dt = 2\pi/ns$$

To see how this works try running the program shown in Fig. 10.8.

This program will draw a series of figures of increasing size centred on a point near the middle of the screen as shown in Fig. 10.9. The figures have an increasing number of sides starting with a triangle.

The program can easily be modified to produce a series of random polygons with different numbers of sides and different sizes. A point to note here is that the

```

100 REM Octagon drawing program
110 LET xc=128
120 LET yc=88
130 LET r=50
140 LET dt=2*PI/8
150 LET th=0
160 LET x1=xc+r
170 LET y1=yc
180 PLOT x1,y1
190 FOR n=1 TO 8
200 LET th=dt*n
210 LET x2=xc+r*COS th
220 LET y2=yc+r*SIN th
230 DRAW x2-x1,y2-y1
240 LET x1=x2
250 LET y1=y2
260 NEXT n

```

Fig. 10.7. Program to draw an octagon.

values of x, y and r should be chosen so that no point of a polygon falls outside the screen limits.

```

100 REM Nested polygons
110 LET r=12
120 LET xc=128
130 LET yc=88
140 FOR k=3 TO 9
145 REM Set number of sides
150 LET ns=k
160 LET dt=2*PI/ns
170 LET x1=xc+r
180 LET y1=yc
190 PLOT x1,y1
195 REM Draw polygon
200 FOR n=1 TO ns
210 LET th=n*dt
220 LET x2=xc+r*COS th
230 LET y2=yc+r*SIN th
240 DRAW x2-x1,y2-y1
250 LET x1=x2
260 LET y1=y2
270 NEXT n
280 LET r=r+12
290 NEXT k

```

Fig. 10.8. Concentric polygons program.

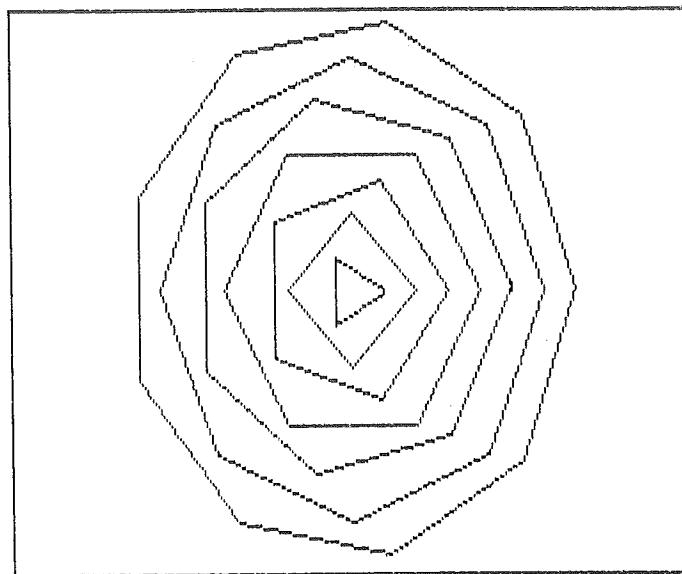


Fig. 10.9. Screen display produced by Fig. 10.8.

Drawing polygons by rotation

Now if we can draw circles using the line rotation technique then it should be possible to draw octagons and polygons as well. For an octagon the angle th will be $2*PI/8$ so a subroutine for drawing an octagon would be as shown in Fig. 10.10.

As before we can develop this little routine into a general polygon drawing routine by adding the variable k (number of sides) and we can produce a program which draws polygons with from 3 to 12 sides in various sizes all over the screen. This is shown in Fig. 10.11.

This procedure for drawing polygons can be used as a general method in any program. Note that it will draw squares and triangles but the triangles will always be equal sided ones.

Star shaped figures and wheels

The polygon drawing routine can easily be modified to draw star shaped figures. In this case a line is drawn from the centre to each calculated point of the polygon by changing the drawing procedure. The program listing in Fig. 10.12 will draw a pattern of random star shaped figures on the screen.

In this case the line is drawn from each corner of the polygon back to the centre so that the lines radiate from the centre like the spokes of a wheel.

Wheel shapes can be drawn by firstly drawing the star and then drawing a circle of the same radius around the same centre point.

```

100 REM Octagon by rotation method
110 LET xc=128
120 LET yc=88
130 LET r=50
140 GO SUB 500
150 STOP
500 REM Octagon drawing subroutine
510 LET x1=r
520 LET y1=0
530 LET th=2*PI/8
540 LET sn=SIN th
550 LET cn=COS th
560 PLOT xc+x1,yc+y1
570 FOR n=1 TO 8
580 LET x2=x1*cn-y1*sn
590 LET y2=x1*sn+y1*cn
600 DRAW x2-x1,y2-y1
610 LET x1=x2
620 LET y1=y2
630 NEXT n
640 RETURN

```

Fig. 10.10. Drawing an octagon by rotation method.

```

100 REM Random polygons
110 FOR j=1 TO 20
120 LET r=10+INT (RND*40)
130 LET xc=r+INT (RND*(255-2*r))
140 LET yc=r+INT (RND*(175-2*r))
150 LET k=3+INT (RND*5)
160 GO SUB 500
170 NEXT j
180 STOP
490 REM Polygon subroutine
500 LET th=2*PI/k
510 LET sn=SIN th
520 LET cn=COS th
530 LET dx=r
540 LET dy=0
550 PLOT xc+dx,yc+dy
560 FOR n=1 TO k
570 LET xr=dx*cn-dy*sn
580 LET yr=dx*sn+dy*cn
590 DRAW xr-dx,yr-dy
600 LET dx=xr
610 LET dy=yr
620 NEXT n
630 RETURN

```

Fig. 10.11. Random polygons program.

```

100 REM Star shaped figures
110 FOR j=1 TO 20
120 LET r=10+INT (RND*40)
130 LET xc=r+INT (RND*(255-2*r))
140 LET yc=r+INT (RND*(175-2*r))
150 LET k=3+INT (RND*5)
160 GO SUB 500
170 NEXT j
180 STOP
490 REM Star shape subroutine
500 LET th=2*PI/k
510 LET sn=SIN th
520 LET cn=COS th
530 LET dx=r
540 LET dy=0
550 FOR n=1 TO k
560 LET xr=dx*cn-dy*sn
570 LET yr=dx*sn+dy*cn
580 PLOT xc,yc
590 DRAW xr,yr
600 LET dx=xr
610 LET dy=yr
620 NEXT n
630 RETURN

```

Fig. 10.12. Program to draw star shaped figures.

Scaling and stretching

In drawing squares, polygons and circles the size of the displayed figure depends upon the value of r that we use in the drawing routine. Thus by altering r we can alter the size or scale of the figure.

In the case of the rectangle there are two scaling figures, one for width (W) and one for height (H). In effect we have a square which has been stretched or compressed in one direction. Assuming that we apply stretching only horizontally or vertically this just means that the x and y scale values are different.

We could apply the stretching idea to other figures by putting in two extra variables which would be the x and y scale factors. To achieve the correct results the reference point around which the figure is drawn should be at the centre of the figure. For polygons and circles this is always true in the drawing methods we have used. In this case the scale factors are used as multipliers for the dx and dy terms in the drawing calculations. Note that the scale factors are not applied to the screen co-ordinates cx, cy around which the figure is drawn.

Let us consider a circle and we will use the trigonometric drawing method as shown in Fig. 10.13. Two new terms sx and sy are now used and the values of dx and dy are multiplied by sx and sy respectively before the figure is drawn. First the

```

100 REM Scaling and stretching
110 REM applied to a circle
120 LET xc=40
130 LET yc=88
140 LET r=40
145 REM Scaling applied to y
150 FOR a=1 TO 0 STEP -.2
160 LET sx=1
170 LET sy=a
180 GO SUB 500
190 NEXT a
200 LET xc=210
205 REM Scaling applied to x
210 FOR a=1 TO 0 STEP -.2
220 LET sx=a
230 LET sy=1
240 GO SUB 500
250 NEXT a
260 LET xc=128
265 REM Scaling of y then x
270 FOR a=1 TO 0 STEP -.2
280 LET sx=1
290 LET sy=a
300 GO SUB 500
310 NEXT a
320 FOR a=1 TO 0 STEP -.2
330 LET sx=a
340 LET sy=1
350 GO SUB 500
360 NEXT a
370 STOP
490 REM Circle subroutine
500 LET dt=2*PI/r
510 LET x1=xc+sx*r
520 LET y1=yc
530 PLOT x1,y1
540 FOR n=1 TO r
550 LET x2=xc+sx*r*COS (n*dt)
560 LET y2=yc+sy*r*SIN (n*dt)
570 DRAW x2-x1,y2-y1
580 LET x1=x2
590 LET y1=y2
600 NEXT n
610 RETURN

```

Fig. 10.13. Demonstration of ellipse drawing.

circle is drawn at the left of the screen with sy being increased in steps from 0 to 1 and sx constant at 1. Next the circle is drawn at the right of the screen with sx increasing from 0 to 1 and sy set at 1. Finally both sx and sy are varied from 0 to 1.

If sx and sy are both 1 then the figure drawn will be a circle. If $sx < 1$ and $sx \leq 1$ the figure becomes an ellipse with the longer axis horizontal. If $sx > 1$ and $sy \geq 1$ the ellipse will have its long axis vertical. If sx or sy is negative this will simply have the effect that the figure is drawn backwards. If we had a figure that was not symmetrical then the left side would be displayed at the right or the top would move to the bottom giving a mirror image effect.

Rotation of figures

The figures we have produced so far have all been drawn with their x axis horizontal. Suppose, however, we want to draw a rectangle but have it displayed tilted at an angle as shown in Fig. 10.14. We have already seen that a point can readily be rotated relative to another point on the screen and this technique was

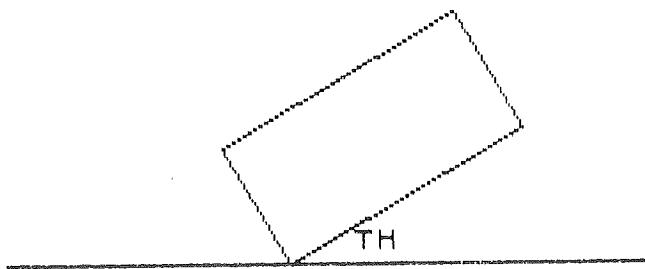


Fig. 10.14. Diagram showing a shape rotated by angle TH.

used for drawing polygons and circles. If we can rotate one point then we can just as easily rotate all of the points in a figure. In this case the rotation equations are applied to each point on the figure in turn to calculate a new point position for the rotated figure. When the figure is drawn using the new set of points it will be tilted relative to the horizontal.

To find the rotated values for DX and DY we use:

$$XR = DX \cdot \cos(TH) - DY \cdot \sin(TH)$$

and

$$YR = DX \cdot \sin(TH) + DY \cdot \cos(TH)$$

where DX and DY are the co-ordinates of each point measured relative to the point about which we want to rotate the figure. The angle of rotation is TH radians relative to the positive x axis.

Let us start with a rectangle and assume that we are using the bottom left-hand corner as a reference point about which the rectangle will be rotated. We shall assume that the rectangle has width W and height H and that the angle through which it is to be rotated is TH.

We can start by setting X_1 , Y_1 to the co-ordinates of the reference point at the bottom left corner which has actual screen co-ordinates XC , YC . The first line to be drawn is the bottom side so the first point to be rotated is at the right bottom corner. For this point the X and Y offsets DX and DY , measured from the bottom left corner which has actual screen co-ordinates XC , YC , are $DX = W$ and $DY = 0$. At this point we can calculate the rotated position of this point (XR , YR) by substituting the values for DX and DY in the rotation equations.

To draw the first line we start by setting X_1 and Y_1 to XC and YC respectively and then plotting a point. Next we calculate the co-ordinates of the other end of the line X_2 , Y_2 by adding the values for XR and YR to the reference co-ordinates XC , YC as follows:

$$\begin{aligned} X_2 &= XC + YR \\ Y_2 &= YC + YR \end{aligned}$$

The first line can be drawn by using the X_1 , Y_1 and X_2 , Y_2 co-ordinates for each end of the line in a DRAW statement as follows:

DRAW $X_2 - X_1, Y_2 - Y_1$

To draw the second side of the rectangle we now set X_1 , Y_1 equal to the values X_2 , Y_2 so that the new line starts from the end of the first. The DX value for the second point is still equal to W , but the DY value is now equal to height H . With these new values we can again apply the rotation equations to obtain new values for XR and YR and for X_2 and Y_2 . Now the second side can be drawn. This process is then repeated for the remaining two sides. Since the rotation and drawing steps are repeated it is convenient to make these into a subroutine and the program for drawing a simple rotated rectangle would be as shown in Fig. 10.15.

The rotation effect when combined with scale changes can produce interesting spiral patterns as shown by the program listed in Fig. 10.16.

We can apply this technique of rotation to any of the figures that can be generated by a series of mathematical steps. When we have an irregular figure however things are a little more difficult. For such figures it is best to make up a table of data points for each of the corners of the figure. These X , Y points are all measured relative to some point on the figure about which it is to be rotated. This may be the centre of the figure or it may be point on the outside of the figure. To draw the figure we simply take the points in turn and draw lines linking them. One further problem arises however. Sometimes we may just want to move to the next point without drawing a line. This can be catered for by producing a third data array which we shall call L . If L is set at 1 a line is drawn and if L is set at 0 no line is drawn.

Having produced the table of X , Y and L values we can now proceed to draw the figure using much the same technique as we did for drawing the rectangle, except that now the X and Y values are taken successively from the arrays. The value for X_1 , Y_1 is initially set at the screen co-ordinates about which we want to draw the shape and then X_2 , Y_2 are calculated using the rotation equations and adding the rotated values to XC and YC respectively. Before the line is drawn L is checked

```

100 REM Rotated rectangle
110 LET xc=128
120 LET yc=40
130 LET w=100
140 LET h=30
150 LET dt=PI/6
160 LET th=0
170 FOR n=1 TO 6
180 LET x1=xc
190 LET y1=yc
200 PLOT x1,y1
210 LET x2=xc+w*COS th
220 LET y2=yc+w*SIN th
230 DRAW x2-x1,y2-y1
240 LET x1=x2
250 LET y1=y2
260 LET x2=xc+w*COS th-h*SIN th
270 LET y2=yc+w*SIN th+h*COS th
280 DRAW x2-x1,y2-y1
290 LET x1=x2
300 LET y1=y2
310 LET x2=xc-h*SIN th
320 LET y2=yc+h*COS th
330 DRAW x2-x1,y2-y1
340 LET x1=x2
350 LET y1=y2
360 LET x2=xc
370 LET y2=yc
380 DRAW x2-x1,y2-y1
390 LET th=th+dt
400 NEXT n

```

Fig. 10.15. Program to draw a series of rotated rectangles.

and if it is 0 the DRAW command is skipped. If no line is drawn the PLOT command for the start point of the next line will move the graphics cursor into position ready to draw the new line. After each line has been processed the values of X_1 , Y_1 are updated to the values of X_2 , Y_2 ready for the next line to be dealt with. This process continues until the figure is complete.

If there are several points in the figure this rotation routine can be speeded up somewhat. Since the angle TH is constant for all points in the figure, the values of $SIN(TH)$ and $COS(TH)$ could be calculated before starting the drawing loop. Now the results of these calculations SN and CN can be used inside the loop thus saving many trigonometric calculations which tend to slow down program execution. The program would then become as shown in Fig. 10.17.

Here, since we have used X and Y arrays to define the points in the figure, the

```

100 REM Patterns using rotation
110 REM and scaling
120 LET th=0
130 LET dt=PI/12
140 LET xc=128
150 LET yc=88
160 FOR w=2 TO 70 STEP 2
170 LET x1=xc
180 LET y1=yc
185 PLOT x1,y1
190 LET dx=w
200 LET dy=0
210 GO SUB 500
220 LET x1=x2
230 LET y1=y2
240 LET dx=w
250 LET dy=w
260 GO SUB 500
270 LET x1=x2
280 LET y1=y2
290 LET dx=0
300 LET dy=w
310 GO SUB 500
320 LET x1=x2
330 LET y1=y2
340 LET dx=0
350 LET dy=0
360 GO SUB 500
370 LET th=th+dt
380 NEXT w
390 STOP
490 REM Rotation subroutine
500 LET x2=xc+dx*COS th-dy*SIN th
510 LET y2=yc+dx*SIN th+dy*COS th
520 DRAW x2-x1,y2-y1
530 RETURN

```

Fig. 10.16. Patterns using rotation and scaling.

variables XC and YC have been used to define the origin point around which the figure will be drawn on the screen.

```

100 REM Rotating an irregular shape
110 DIM x(5)
120 DIM y(5)
130 DIM l(5)
135 REM Set up data for figure
140 FOR n=1 TO 5
150 READ x(n),y(n),l(n)
160 NEXT n
170 DATA 20,0,0
180 DATA 60,0,1
190 DATA 60,-15,0
200 DATA 40,0,1
210 DATA 60,15,1
220 LET xc=128
230 LET yc=88
240 LET dt=2*PI/10
250 LET th=0
260 FOR f=1 TO 10
270 LET sn=SIN th
280 LET cn=COS th
290 LET x1=xc
300 LET y1=yc
310 GO SUB 500
320 LET th=th+dt
330 NEXT f
340 STOP
490 REM Figure drawing subroutine
500 PLOT x1,y1
510 FOR n=1 TO 5
520 LET x2=xc+x(n)*cn-y(n)*sn
530 LET y2=yc+x(n)*sn+y(n)*cn
540 IF l(n)=1 THEN DRAW x2-x1,y2-y1
550 PLOT x2,y2
560 LET x1=x2
570 LET y1=y2
580 NEXT n
590 RETURN

```

Fig. 10.17. Rotation of an irregular shape.

Chapter Eleven

Making the Most of High Resolution

This short chapter should start you on your way to using high resolution graphics to their full advantage: and the first thing to think about is how you can combine low and high resolution graphics on the same screen. In practice it's simple enough – the Spectrum is quite happy to accept PLOT and DRAW commands together with PRINT AT commands in the same program. But suppose PRINT AT doesn't place your text precisely where you want it? There will be times when you want to place a symbol at a specific point on the screen which may lie between the normal print positions. This can be achieved fairly easily and we shall now examine the way in which it is done.

Remember that a symbol simply consists of an array of dots and if we use the PLOT command dots can readily be set up at any point on the screen. Let us suppose we want to take the symbol A and place it at some random point on the screen. The first thing we need to know is the pattern of dots that make up the A symbol. The easiest way to get the dot pattern is to print the A at position $0,0$ on the screen. We now know the exact position of this pattern of dots and we can use the POINT command to discover which dots are in INK colour and which are PAPER colour. By using a simple loop operation we can examine each dot of the printed symbol at a time and then we can print a copy of it at any desired point on the screen. The program listed in Fig. 11.1 shows how the A symbol can be printed at a point near the centre of the screen using this technique.

The required symbol is first printed at the top left corner of the screen to give the dot pattern that we are going to copy. The upper row of dots in this pattern has x

```
100 REM Placing a symbol at the
110 REM screen centre by dot copy
120 PRINT AT 0,0;"A";
130 LET x=128
140 LET y=88
150 REM Copy dot pattern
160 FOR b=0 TO 7
170 FOR a=0 TO 7
180 IF POINT (a,175-b)=0 THEN GO TO 200
190 PLOT x+a,y-b
200 NEXT a
210 NEXT b
```

Fig. 11.1. Transfer of a symbol by dot copying.

locations running from 0 to 7 and their y position is 175. The next row of dots also has the same x positions but y is now 174 and successive rows are the same except that y is reduced by 1 for each row. To scan the dots we can set up two count loops with variables a and b which both run from 0 to 7.

For the POINT command the x and y parameters will be a and 175-b and we simply have to check if the result is 1 or not to see if the dot is in INK colour. Having examined the dot pattern we now have to plot a copy of the dot pattern at some other desired point on the screen. First we must define the point at which we want to plot the symbol and here it is convenient to select the x,y co-ordinates as the point where the top left corner of the displayed symbol is to be. Now to plot the points we simply use PLOT x+a,y-b where a and b are the same count values as we used in the POINT command. If the POINT test shows a lit dot then a dot is plotted in the new symbol position, but if the dot is off, the PLOT instruction is skipped.

As the two loops progress the dot pattern will be copied from the top left corner to the new position. Using this routine it is possible to plot two symbols overlapping quite easily. This is demonstrated in the program listed in Fig. 11.2 where the symbol is copied to random positions around the screen. This produces

```

100 REM Placing a symbol at random
110 REM positions by dot copying
120 PRINT AT 0,0;"A";
130 FOR n=1 TO 50
140 LET x=8+INT (RND*240)
150 LET y=8+INT (RND*160)
160 GO SUB 400
170 NEXT n
180 STOP
390 REM Copy symbol to point x,y
400 FOR b=0 TO 7
410 FOR a=0 TO 7
420 IF POINT (a,175-b)=0 THEN GO TO 440
430 PLOT x+a,y-b
440 NEXT a
450 NEXT b
460 RETURN

```

Fig. 11.2. Transferring symbols to random screen positions.

a screen display similar to that shown in Fig. 11.3. This method of setting up text symbols on the screen is particularly useful when text has to be inserted into a high resolution graphics drawing.

Rotating the symbols

By slightly changing the copying loop we can now produce some more interesting

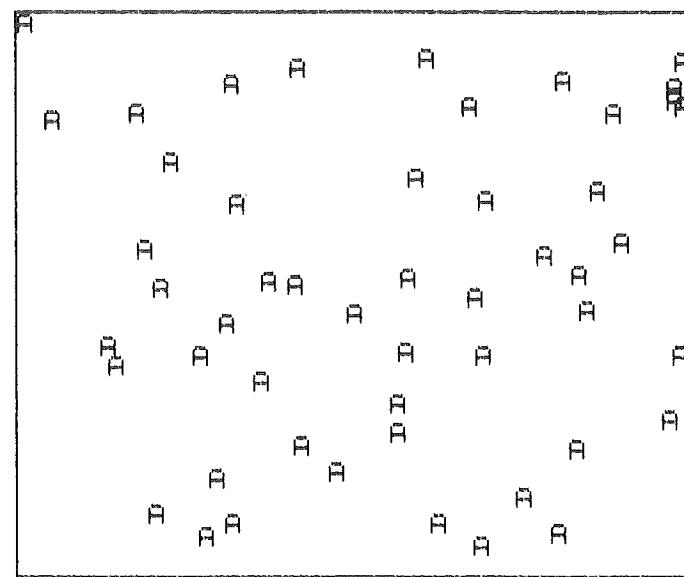


Fig. 11.3. Typical random position symbol display.

effects. Suppose we want to write the symbol upside down on the screen. If we change the y term of the PLOT command to y+b this effectively inverts the symbol, since the top row of dots that was read from the master pattern now becomes the bottom row of dots in the plotted symbol and the sequence of all other rows also changes. An important point to note here is that the plotted symbol will now have its lower left corner at the specified x,y position. The position x,y must be chosen so that the y co-ordinate of the top row of the plotted symbol does not go beyond 175, otherwise the program will stop on an error.

If you want the symbol upside down but with its top left corner still at x,y then the PLOT command must be changed to:

PLOT x+a,y-8+b

This would allow the symbol to be inserted easily into a row of printed text symbols.

To turn a symbol back to front so that it looks like a mirror image, we can apply a similar process to the x term of the PLOT command so that it becomes:

PLOT x+8-a,y-b

Now let us get a little more adventurous and see what happens if we swap the offset terms a and b in the PLOT command to give:

PLOT x+b,y+a

This produces a symbol which has been turned on its side since the horizontal rows of the original pattern have now been plotted vertically. To turn the symbol over to its other side we simply have to change the command to:

PLOT x-b,y+a

```

100 REM Rotation of text symbols
110 REM using dot copying
120 PRINT AT 0,0;"A";
130 LET x=124
140 LET y=96
150 GO SUB 400
160 LET x=124
170 LET y=64
180 GO SUB 500
190 LET x=112
200 LET y=76
210 GO SUB 600
220 LET x=144
230 LET y=84
240 GO SUB 700
250 PRINT AT 0,0;" ";
260 STOP
390 REM Normal symbol
400 FOR b=0 TO 7
410 FOR a=0 TO 7
420 IF POINT (a,175-b)=0 THEN GO TO 440
430 PLOT x+a,y-b
440 NEXT a
450 NEXT b
460 RETURN
490 REM Inverted symbol
500 FOR b=0 TO 7
510 FOR a=0 TO 7
520 IF POINT (a,175-b)=0 THEN GO TO 540
530 PLOT x+a,y+b
540 NEXT a
550 NEXT b
560 RETURN
590 REM Symbol rotated to left
600 FOR b=0 TO 7
610 FOR a=0 TO 7
620 IF POINT (a,175-b)=0 THEN GO TO 640
630 PLOT x+b,y+a
640 NEXT a
650 NEXT b
660 RETURN
690 REM Symbol rotated to right
700 FOR b=0 TO 7
710 FOR a=0 TO 7
720 IF POINT (a,175-b)=0 THEN GO TO 740
730 PLOT x-b,y-a
740 NEXT a
750 NEXT b
760 RETURN

```

Fig. 11.4. Rotation of symbols by dot copying.

Once again in these commands a correction may be made to the basic x,y values so that the top left corner of the new symbol will still be positioned at point x,y on the screen.

The program listed in Fig. 11.4 draws symbols in all four orientations and will produce a display similar to that shown in Fig. 11.5.

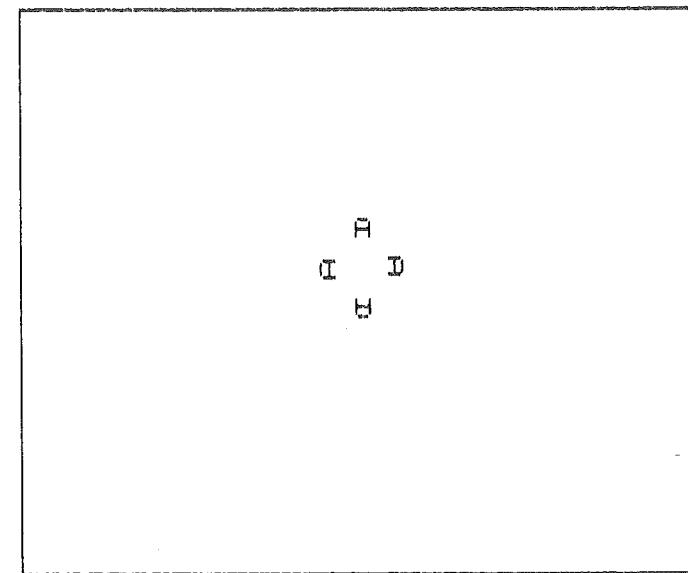


Fig. 11.5. Display of rotated symbols.

Bigger and better characters

Suppose we want to produce a double width symbol on the screen. To get double width we can simply use a second PLOT command to place a dot alongside the first. Of course we must also have two blank spaces for each blank dot. This can readily be achieved by using an offset of twice a so that the plot action moves two points for each point in the original pattern.

It is equally easy to generate double height symbols. In this case the doubling up process is applied to the b offset instead of the a offset. Now each row of dots in the original pattern is scanned twice and produces two rows of dots one above the other in the plotted copy character.

By combining the two actions we can produce double size symbols. Further development along these lines will allow treble or quadruple size symbols to be produced from the original dot pattern. An important point to watch when using any of the dot pattern copying routines is to make sure that none of the dots are allowed to go beyond the screen limits.

The program listed in Fig. 11.6 gives some idea of the possibilities of this technique and shows a range of symbols in sizes up to five times as large as the

```

100 REM Producing large symbols
110 LET cy=165
115 REM v=symbol height
120 FOR v=1 TO 5
130 LET cx=10
140 LET cy=cy-8*v
145 REM h=symbol width
150 FOR h=1 TO 5
160 PRINT AT 0,0;"$";
170 LET cx=cx+10*h
180 GO SUB 500
190 NEXT h
200 NEXT v
220 PRINT AT 0,0;" ";
230 STOP
490 REM Symbol copying subroutine
500 FOR y=0 TO 7
510 FOR x=0 TO 7
520 IF POINT (x,175-y)=0 THEN GO TO 580
530 FOR k=0 TO v-1
540 FOR j=0 TO h-1
550 PLOT cx+h*x+j,cy-v*y+k
560 NEXT j
570 NEXT k
580 NEXT x
590 NEXT y
600 RETURN

```

Fig. 11.6. Program to produce bigger symbols.

original character. The display produced on the screen by this program is shown in Fig. 11.7. This technique can be very useful for providing bold titles on the screen display.

Sloping symbols

Sometimes we may want to produce italic style symbols where the displayed symbol is inclined at an angle instead of being vertical. This result can be obtained by subtracting the b offset from the combined a and x term. Now each successive row of dots in the symbol is displaced one position to the left so that the symbol is drawn at an angle and looks very much like a rather exaggerated italic symbol. A more realistic looking italic symbol is produced by using INT (b/2) instead of b, since this reduces the slope of the symbol.

If instead of subtracting the b term it is added to the x term then the symbols will

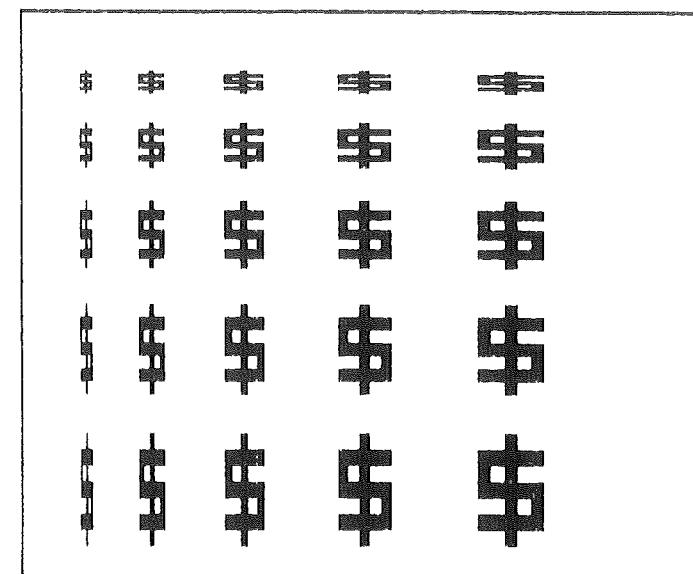


Fig. 11.7. Big symbol display.

```

100 REM Large italic symbols
110 LET cy=165
115 REM Set symbol height (v)
120 FOR v=1 TO 5
130 LET cx=10
140 LET cy=cy-8*v
145 REM Set symbol width (h)
150 FOR h=1 TO 5
160 PRINT AT 0,0;"$";
170 LET cx=cx+10*h
180 GO SUB 500
190 NEXT h
200 NEXT v
210 PRINT AT 0,0;" ";
220 STOP
490 REM Symbol copy subroutine
500 FOR y=0 TO 7
510 FOR x=0 TO 7
520 IF POINT (x,175-y)=0 THEN GO TO 580
530 FOR k=0 TO v-1
540 FOR j=0 TO h-1
550 PLOT cx+h*x+j-y*h/2,cy-v*y+k
560 NEXT j
570 NEXT k
580 NEXT x
590 NEXT y
600 RETURN

```

Fig. 11.8. Creating italic-style symbols.

slope in the opposite direction. The results of this type of operation can be seen by running the program listed in Fig. 11.8. The results produced on the screen are shown in Fig. 11.9, which demonstrates the effect produced on a variety of symbol sizes and shapes.

Other possibilities with which you might experiment are to add the a offset to the y term, which will give a character with sloping horizontal lines, or to combine both operations, which will draw the character rotated through 45 degrees. Since this rotation technique does not follow the normal rotation equations, the shape of the symbol will be distorted when it is rotated in this way. I will leave you to experiment with the various possible combinations that can be applied to the plotting of the copied character.

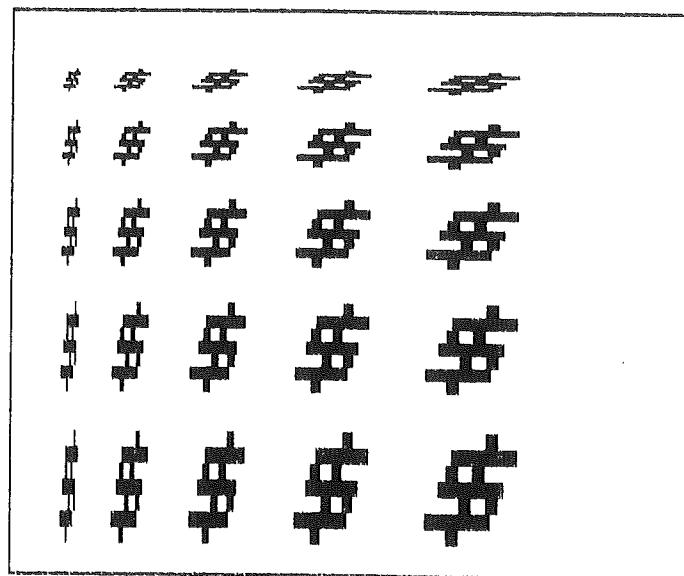


Fig. 11.9. Display of italic symbols.

Making a sketching program

So far we have used PLOT and DRAW commands with points worked out either by calculation or by the computer itself. This isn't the way you would draw a picture! As a final piece of Spectrum graphics magic, let's take a look at a program that will allow you to treat your Spectrum like a computerised sketchpad.

In this program the arrow keys can be used to move the graphics cursor round the screen. With a little modification (see Part 5) you could also use a joystick and a suitable interface to do this. Let's assume that the graphics cursor is the tip of a pen, and the pen can either be on the paper (down) or lifted off the paper (up). When the pen is down it will leave a trace on the screen, when it's up it will simply

```

100 REM Sketching program
110 PRINT "Arrow keys move pen up/down left/right"
120 PRINT "D key puts pen down for drawing"
130 PRINT "U key lifts pen for moving"
140 PRINT "E key erases line"
150 PRINT "Number keys 0 to 6 set colour"
160 PRINT "Keys C,V,B and N give diagonal lines"
170 INPUT "Ready? (Y/N) ";a$
180 IF a$<>"y" THEN GO TO 170
185 REM Initialise
190 LET x=0: LET y=0: LET x1=0: LET y1=0
200 LET s=0: LET p=0: INK 0: PAPER 7
210 LET e=0: CLS : PLOT x,y
215 REM Test for key pressed
220 IF INKEY$="" THEN GO TO 220
230 LET a$=INKEY$
235 REM Check for quit
240 IF a$="q" OR a$="Q" THEN STOP
245 REM Check required pen state
250 IF a$="u" OR a$="U" THEN LET p=0: GO TO 270
260 IF a$="d" OR a$="D" THEN LET p=1: LET e=0
270 IF a$="e" OR a$="E" THEN LET e=1
275 REM Check for arrow keys and set new x,y
280 IF a$=CHR$ 8 THEN LET x=x-1: GO TO 360
290 IF a$=CHR$ 9 THEN LET x=x+1: GO TO 360
300 IF a$=CHR$ 10 THEN LET y=y-1: GO TO 360
310 IF a$=CHR$ 11 THEN LET y=y+1: GO TO 360
315 REM Check for and set diagonal moves
320 IF a$="c" OR a$="C" THEN LET x=x-1: LET y=y+1
330 IF a$="v" OR a$="V" THEN LET x=x-1: LET y=y-1
340 IF a$="b" OR a$="B" THEN LET x=x+1: LET y=y-1
350 IF a$="n" OR a$="N" THEN LET x=x+1: LET y=y+1
355 REM Check x,y limits and execute wraparound
360 IF x>255 THEN LET x=x-256
370 IF x<0 THEN LET x=x+256
380 IF y>163 THEN LET y=y-164
390 IF y<0 THEN LET y=y+164
395 REM Set up colour
400 LET c=(CODE a$)-48
410 IF c>6 OR c<0 THEN GO TO 430
420 INK c
425 REM Replace previous state if pen is up
430 IF p=1 OR s=1 THEN GO TO 450
440 PLOT OVER 1;x1,y1: OVER 0
450 LET s=POINT (x,y)
455 REM Carry out erasure
460 IF e=1 THEN INVERSE 1
465 REM Plot new point
470 PLOT x,y
475 REM Update last point marker
480 LET x1=x: LET y1=y
485 REM Print current x,y position
490 PRINT AT 0,0;"x = ";x;" y = ";y;" "
500 INVERSE 0
510 GO TO 220

```

Fig. 11.10. High resolution sketching program.

move to a new position. To control this you can use the U (for up) and D (for down) keys. At the start of the program, listed in Fig. 11.10, the pen is 'up' and is at the bottom left-hand corner of the screen, where $X=0$ and $Y=0$.

There's a problem, though. When the pen is up nothing is drawn, so we have no way of knowing where the pen is. Here the problem is overcome by placing a dot on the screen at the point where the pen is placed. Each time the pen moves the dot is erased, and then redrawn in a new position so that it continually shows where the pen is at any time. If there is already a lit dot at the position before the pen moves to it this is noted by using the POINT command and the state of the pixel point is saved. When the pen moves to a new position the original state of the pixel is restored. If this were not done then passing the pen over a line already drawn on the screen could cause part of that line to be erased.

The arrow keys and the U and D keys are continuously monitored by using a loop and the INKEY\$ command. When no key is pressed INKEY\$ returns a blank string ("") and the test is repeated again. When a key is pressed a\$ is set to INKEY\$ and then is tested to see which key has been pressed and the appropriate action is taken. For the arrow keys to work correctly the CAPS SHIFT key must be held down whilst the arrow key is pressed. The program detects the actual code for the shifted key. If an arrow key is held down the auto-repeat action of the Sinclair keyboard will come into play and a series of steps is drawn in succession to produce a line.

The program contains some further refinements. If the E key is pressed the pen will act as an eraser and will blank out any points that it passes over. This will allow the user to correct mistakes. Using the arrow keys will allow only horizontal or vertical lines to be drawn. Four extra keys are also recognised by the program. These are the C, V, B and N keys which are programmed to give diagonal movement to the pen. The N key moves up and to the right whilst the B key moves down to the right. The C and V keys move to the left and up and down respectively.

Chapter Twelve

The Sound of Spectrum

The sound generating capabilities of the Spectrum are modest, but useful. There is no way of controlling amplitude (loudness) or waveform (the timbre of a note, so that it can simulate instruments), and the only quantities that we can alter are the duration of each note and its frequency (pitch). These are altered by the numbers which must follow the BEEP instruction, which is the one that we use for generating sounds. BEEP is a single-channel instruction - you cannot produce two notes (or more) sounding together - and on the built-in loudspeaker (more like a quietspeaker) it produces sounds that you may have to strain to hear, especially if you are deaf through attending discos.

The built-in chirp-maker is used, of course, to produce the click that you hear each time you press a letter/number key, and is controlled by a location 236\$9. This is alterable memory (RAM) so that we can alter the duration of the click, making it more of a cheep, by a command entered directly (no line number):

POKE 236\$9,1\$0 (press ENTER)

This effect lasts for as long as the computer is switched on or until you put a different value in this memory address. The original number at address 236\$9 is \$0; this is the value that is put into this location when the computer is first switched on, and the action of putting it there is controlled by a program in unalterable memory (ROM) so that it happens each and every time.

The sounds that are produced by the BEEP instruction are put out as electrical signals on both of the cassette leads. If you keep the MIC lead permanently plugged in to the precious recorder that you reserve for saving and loading programs, you can use the EAR lead to carry these sound signals to an amplifier so that you can hear the sounds at a much higher volume than can be attained from the built-in speaker. An alternative which is possible with some cassette recorders is to press RECORD, with no cassette in place (you will have to press in the lever at the back left-hand side of the recorder, where the cassette fits), and PLAY, and then use an external loudspeaker connected to the EAR socket. Not all cassette recorders will work in this way, because many of them use the same circuits for recording as for replaying, and can't operate a loudspeaker while they are recording. A third possibility is to buy a jack socket-to-socket connector, and plug the EAR plug into it, then a set of headphones, so linking the headphones to the EAR socket. This gives private listening, but you have no control over the volume.

BEEP, as we have said, needs two numbers following the instruction; the first number controls duration and the second controls frequency. Duration is

expressed in seconds, which makes life reasonably easy, and the number need not be a whole number; items such as 1.25 will be acceptable. Frequency is a number based on a scale which uses both positive and negative numbers. Now at this point, anyone with any faint memories of learning music has a distinct advantage, and can skip parts of this, but the non-musical will need some help, because we have to relate the frequency number to the musical scale, and if you don't know the musical scale, you haven't much to relate to!

Keyboard instruments, of which the best known is the piano, use a set of keys to obtain notes of the musical scale; one key, one note. You can't have any notes that are half-way between the notes that are obtained from neighbouring keys (unless you re-tune the thing between notes!), and the keyboard is arranged with white keys and black keys. A black key gives a note that is half-way in pitch between the notes from its neighbouring white keys. Instruments which use strings without keyboards can produce notes which the piano can't, such as one lying between a piano 'black' note and its neighbouring 'white' note. Orchestras have the problem of ensuring that all of the instruments play the same scale of notes, and so they all tune to a note from one instrument, usually an oboe, which has in turn been set up from a standard tuning-fork.

Spectrum is tuned rather like a keyboard instrument, but with the 'half-way' notes (between piano black and white keys) available. It takes as its standard of pitch the note that is at the centre of the piano keyboard, called Middle C. Its frequency number as used in Spectrum is \emptyset (lower notes use negative numbers), but we know that the note on keyboard instruments actually corresponds to 261.6 vibrations of air per second. The program in Fig. 12.1 gives you some idea of what

```

10 CLS : FOR K=1 TO 8
20 READ n$,note: PRINT AT 10,2
*K:n$#
30 BEEP 1,note: NEXT K
40 PRINT "All done."
50 GO TO 9999
60 DATA "C",0,"D",2,"E",4,"F",
5,"G",7,"A",9,"B",11,"C",12

```

Fig. 12.1. Demonstrating the BEEP instruction.

these notes sound like by playing a scale slowly and displaying the names of the piano notes. Keeping to whole numbers like this ensures that the notes which you use are piano notes, but there's no reason why you should not use numbers which include fractions, like 1.5. You can produce a warbling note or trill by going rapidly between a BEEP at one note and one which is 0.5 different in pitch, for example. These warbles, as illustrated in Fig. 12.2, are usually more attention-getting than simple straightforward notes, and they make good warning devices, particularly at high pitches. In the program of Fig. 12.2, the duration of the warble has had to be set by a loop, because the duration number in the BEEP instruction has been used to set the time of each individual note in the warble.

Though we can't produce harmony with a single channel of sound, we can create a faint illusion of harmony by warbling between notes which if played

```

10 CLS : PRINT TAB 15;"Trills"
FOR n=1 TO 4: READ j,j
20 FOR k=1 TO 50: BEEP .01,j:
BEEP .01,j: NEXT k
30 NEXT n: GO TO 9999
40 DATA 0,.5,2,2.5,5,5.5,6,6.5

```

Fig. 12.2. A warbling note program.

```

10 CLS : PRINT TAB 12;"Harmony"
": FOR n=1 TO 4: READ j,j
20 FOR k=1 TO 50: BEEP .02,j:
BEEP .02,j: NEXT k
30 NEXT n
40 DATA 0,4,5,11,9,14,12,16

```

Fig. 12.3. Producing an effect of harmony.

together would produce harmony. As a method, it was good enough for Johann Sebastian Bach, so it ought to be good enough for Spectrum. Figure 12.3 illustrates some 'harmony' produced by this method – the time duration needs to be fairly short to make it sound convincing.

We can tackle rising or falling notes in the same way, by setting up a BEEP instruction whose pitch number is the counter of a FOR ... NEXT loop which in the example of Fig. 12.4 has a STEP of 0.5. Shorter STEP intervals can be used to

```

10 CLS : PRINT TAB 14;"Wait"
FOR n=1 TO 5
20 FOR j=0 TO 18 STEP .5: BEEP
.01,j: NEXT j: REM upward slide
30 NEXT n

```

Fig. 12.4. A wail whose pitch ascends.

extend the slide – obviously if we had written the loop as FOR j=18 TO \emptyset STEP $-.5$, the slide of notes would have been downwards rather than upwards.

Following a score

The BEEP facility allows you to 'write' music (of a limited kind) into the computer from a musical score. If the part that you are following is a piano part, then writing the BEEP commands should be straightforward, using the diagram of Fig. 12.5 as a guide to translate the musical notes into BEEP pitch and duration numbers. If you know nothing about music, then you will have to learn how the marks \sharp (sharp – don't omit the s!) and \flat (flat) are used. The \sharp sign means take the next BEEP number up from the normal note at that position, working with whole numbers. For example, the note C has the BEEP number \emptyset , so that C \sharp (c-sharp) would have the BEEP number 1. Similarly, since D has the BEEP number 2, D \flat (d-flat) also has the BEEP number of 1. The fact that these two are identical is a peculiarity of the piano scale, and there are instruments on which the two need not be the same, but in which they are often made the same so as to agree with the piano scale.

If the music which you are using is not piano music, you can reasonably use violin scores, but you have to be careful about using music written for some other instruments. Of the instruments that are popular in schools, for example, music for the clarinet has to be translated into Spectrum BEEPs with some caution, because the clarinet is a 'transposing' instrument. This means that when a clarinet player sees middle C on a score (BEEP 0) and places his/her fingers in the correct places, the note that sounds is NOT middle C. This has been done deliberately to make it easier to write music for orchestras, and for musicians to play several different instruments, but it will cause you some confusion if you are using clarinet music as a guide to writing BEEP music. Most clarinets sound B₁ when the music reads C, which corresponds to subtracting 2 from the BEEP pitch number.

Timing

The relative value of time of notes on a musical score is indicated by their shape, and Fig. 12.5 also shows these times. The counts are for fairly slow music, and for

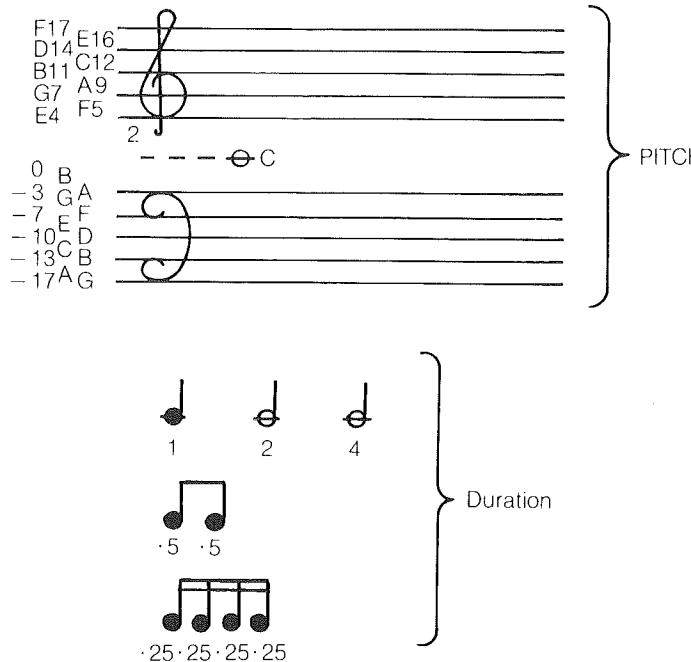


Fig. 12.5. Translating from music into BEEP numbers.

faster pieces you would need to use half or quarter of these values, but keeping the relationships the same. For more details on how to follow a musical score you will have to consult a text on music (which won't tell you anything about computing!).

Finally, the program in Fig. 12.6 allows the Spectrum to simulate a sort of keyboard instrument by producing a note for each letter key. This is a very simple program, which searches through one string to find a match for the key that has

```

5 LET a$="qawsedrftgyhujikolp
": LET b$="-6-5-4-3-2-1+0+1+2+3+
4+5+6+7+8+9101112"
10 CLS : PRINT TAB 14:"Spusic"
: PAUSE 150
20 PRINT """This allows you to
use your""Spectrum as a music
keyboard. """The two middle rows
of keys,"""except for ENTER, ar
e used as""note generators, and
the note" ""will repeat for as
long as ""the key is pressed"
30 PAUSE 250: CLS : PRINT """
NOW
40 LET k$=INKEY$: IF k$="" OR
k$=CHR$ 13 THEN GO TO 40
50 FOR j=1 TO 19: IF k$<>a$(j)
THEN NEXT j
60 LET x=2*j-1: LET y=2*j: LET
q=VAL b$(x TO y)
70 BEEP .25,q: GO TO 40

```

Fig. 12.6. Producing BEEPs from the keyboard!

been pressed, and then locates a pitch value in another string, sounds the note and looks around for another key. You will notice that the time between repeats (when a key is held down) is longer when the note is higher – this is because the note is at the end of the string and so takes longer to find – I did say it was a simple program! Another point to notice is that while the BEEP note is sounding, all other computing activities grind to a halt, so that when you use BEEP in a program that does other activities, its effect is very much the same as that of a PAUSE as far as the timing of the program is concerned.

Part 3

The Leisure Section

*by Vince Apps, Mike James,
S.M. Gee and Kay Ewbank*

The Leisure Section

This section of *The Complete Spectrum* is devoted to the gentle art of enjoying yourself, – and if you've just worked your way through some of the tougher programs and ideas in Part 2 then you may be in need of some relaxation!

These two chapters contain the complete listing for fifteen games. If you like, just enter them (*carefully!*) and see how you enjoy them, but take the chance to study the programming techniques, too, because many of the routines in these games show how the ideas in Parts 1 and 2 can be used in practice. To help older users, each game in Chapter 14 includes an outline of the program structure, but the games in Chapter 13 are designed especially for younger Spectrum users. They're challenging, they're fun, and they're educational, in more ways than one. Besides learning about numbers, words, languages, and constellations, your children will learn a lot about the computer by entering and using them. There's a short introduction to help junior programmers at the beginning of Chapter 13, and once they've worked their way through the chapter they may like to take a crack at some of the more advanced programs in Chapter 14. In these, especially, you'll see the graphics routines from Part 2 in action, and you'll also see how you can animate user-defined characters and other graphic designs.

To give you an idea of what each game is like, a sample screen display is included at the head of each program listing. All the programs in this section were fully tested, in their present form, before publication, and the listings from the operating program were then printed out on a full size printer (see Part 5). When you are typing in these programs for yourself, remember that you will not need to leave spaces after keywords – the Spectrum does that for you. However, there's no such thing as the last bug in a program, so it's possible that even now some mistakes still remain. All the same, if a program doesn't work when you've typed it in, take a break and then check it very carefully again. It's all too easy to read what you think should be there rather than what really is. Particular points to look out for are null strings and blank spaces – have you typed as many of them as you should have done? In Chapter 14 you'll find REM statements telling you how many spaces are needed when it's difficult to see for yourself. Other sources of possible error are the 'greater than' (>) and 'less than' (<) symbols – getting them the wrong way round will cause chaos! And don't be tempted to change the line numbering as you type in the program, as there are far too many pitfalls involved!

All the programs in Chapter 13 are fairly short, but some of those in Chapter 14 are rather longer. Long programs can be a chore to type in, so if you don't want to do it yourself, the games from Chapter 14 are also available as cassette tapes. For full details and an order form send a stamped, self-addressed envelope to RAMSOFT, P.O. Box 6, Richmond, North Yorkshire, DL10 4HL.

Chapter Thirteen

Games for Younger Users

This collection of programs has been written for junior users to pit their wits and test their knowledge. We have designed these programs so you can learn how to use your Spectrum computer by typing in your own information. None of the programs is too long to input into your machine as the idea is to help you sharpen up your mind – not to turn you into a typist.

Once the program is in the machine you will be able to begin answering the questions and improving your knowledge. The faster you respond the quicker the computer will ask you the next question.

You will be surprised how quickly you will learn how to use your Spectrum, and how soon you will want to move on to the next program.

As soon as you know how to enter these programs – and beat the Spectrum – you will be able to change the content in the lists and make things even more difficult for yourself.

Things to remember

Your Spectrum works with its own language called BASIC. If you try to 'speak' to your machine in another language then nothing will happen, except that you will get an 'error message' at the bottom of the screen.

Programming is not like writing an essay for class. Your teacher might let you off with a mild caution if you miss out a comma – your Spectrum will not.

You have to follow, *exactly*, the 'characters' shown on the program listings in this book. If you miss a comma, or enter a dash by error, then the program will not work. You cannot put in any other instruction and expect the machine to work. If you have typed in your program and the Spectrum will not 'run' as instructed, then you will have to check your 'list' against the book for 'bugs'. Check carefully before you decide to throw your machine out of the nearest window!

We know that the programs listed in the book are 'bug free' because, not only have we checked and double-checked them, we have reproduced them from our Spectrum using a mechanical printer.

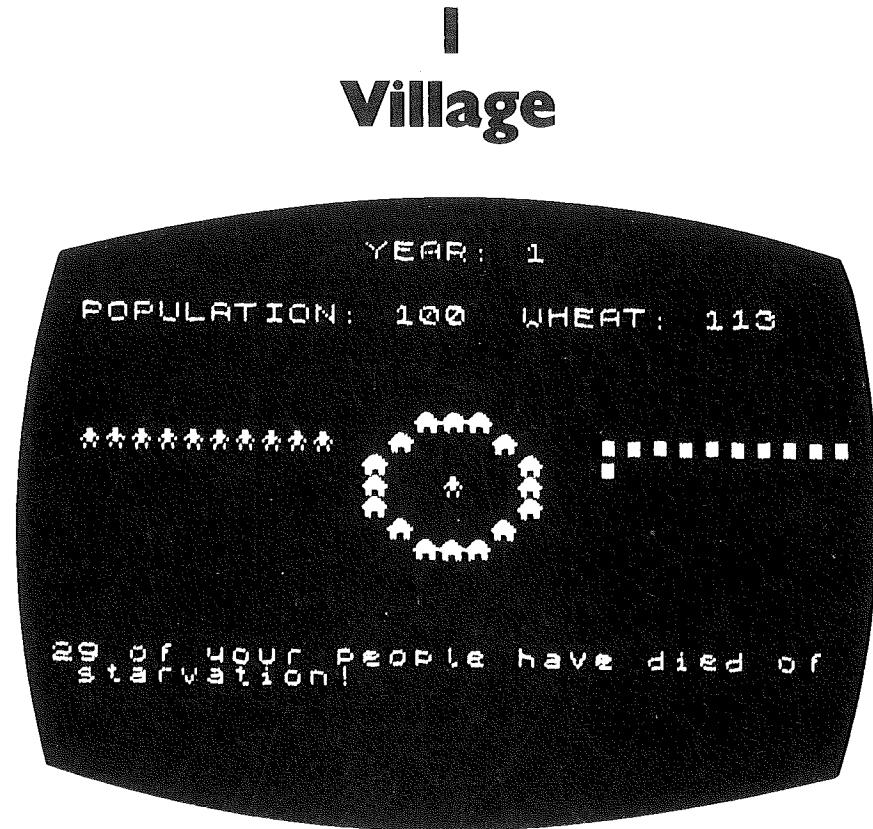
We did this because it makes things easier for you and also for our own printers when they produce our books. If you are really having trouble however you can always ask your parents. They **should** be able to help you.

Cassette storage

Once you have gone through 'inputting' your programs you can store them on cassette tapes for future use.

Cassette tapes will take up some of your pocket money but they mean that you will always have quick access to your list of programs.

Having your programs on tape also means that when you have improved your programming skills, you will be able to rewrite the programs we have given you.



You have just been appointed the chief of a village of natives whose lives depend on their crops of wheat. If you manage the crops properly then the village will prosper and the population will increase but should you make a mistake then people will starve, people will die and you will be attacked by an angry mob.

By the way, your people need about $2 \frac{1}{2}$ bags of wheat each to survive for a year. Give them less and they starve and they won't like that. If you give them 5 bags each they will be pleased and may forgive you for past mistakes.

Look out as well for the rats which always attack your crops in the warehouse. The more you store, the greater will be the losses from rats.

How to play

The screen will show you that you are in your first year as leader. You will begin with a certain population and a certain amount of wheat. There are symbols for people and sacks of wheat and each symbol represents ten units. The computer will ask you how much of your wheat you wish to sow. Remember to keep some back as it might be a bad harvest.

The computer will then tell you how much wheat you have grown and you will be asked how much you wish to give your people.

Try and survive ten years as leader.

Programming notes

The square in line 890 is a graphics symbol – hold down CAPS SHIFT while you press 9, then 8, then 9 again.

Program

```

10 REM VILLAGE
30 RANDOMIZE
50 FOR N=0 TO 31
60 READ D
70 POKE USR "A"+N,D
80 NEXT N
90 DATA 24,60,126,255,126,102,
102,102
100 DATA 24,24,60,90,24,36,36,1
02
110 DATA 0,0,60,60,60,60,60,0
120 DATA 28,18,57,57,56,56,56,1
6
130 LET SF=0
140 LET POP=100
150 LET WHT=250
160 LET YR=1
170 LET RNG=0
180 LET RE=2.4
190 GO SUB 900
200 GO SUB 1000
210 REM GRAPHICS "A" BELOW
220 PRINT INK 4,AT 8,14;"aaa"
230 PRINT INK 4,AT 9,13;"a a "
"240 PRINT INK 4,AT 10,12;"a
"250 PRINT INK 4,AT 11,12;"a
"260 PRINT INK 4,AT 12,12;"a
"270 PRINT INK 4,AT 13,13;"a
"280 PRINT INK 4,AT 14,14;"aaa"

```

```

290 PRINT INK 2,AT 11,15,CHR$145
294 GO SUB 300
296 GO TO 330
300 PRINT PAPER 6, INK 9,AT 3,
1,"POPULATION: ",POP," "
306 LET WHT=INT (WHT+0.5)
310 PRINT PAPER 6, INK 9,AT 3,
18,"WHEAT: ",WHT," "
320 PRINT PAPER 6, INK 9,AT 0,
12,"YEAR: ",YR
322 RETURN
330 LET HVS=INT (RND*3)+1
332 IF HVS=1 THEN LET A$="Poor
"
334 IF HVS=2 THEN LET A$="fair
"
336 IF HVS=3 THEN LET A$="Good
"
338 LET M$="The witch-doctor pr
edicts a "+A$+" harvest.": GO SUB
1100
340 INPUT PAPER 3, INK 9,AT 0,
0;"How much seed will you sow? "
,SD
344 IF SD<0 THEN BEEP 0.9,-12
GO TO 340
350 IF SD>WHT THEN BEEP 0.6,-1
2: LET M$="You do not have "+STR
$ SD+" bags!": GO SUB 1100: GO T
O 340
360 IF SD>POP THEN BEEP 0.6,-1
2: LET M$="There are not enough
people to sow "+STR$ SD+" bags!
": GO SUB 1100: GO TO 340
370 LET CRP=INT (RND*2*HVS*SD)
372 LET WHT=WHT-SD
374 GO SUB 300: GO SUB 1000
376 PAUSE 250
380 LET M$="Your crop was "+STR
$ CRP+" bags of wheat.": GO SUB
1100
384 IF (HVS=2 OR HVS=3) AND CRP
<1.5*SD THEN LET M$="Even witch
-doctors can be wrong!": GO SUB
1100

```

```

400 LET WHT=WHT+CRF
404 GO SUB 300
408 GO SUB 1000
409 PAUSE 250
410 INPUT PAPER 3; INK 9;AT 0,
0;"How much wheat will you give
to your People? ";ET
411 IF ET<0 THEN BEEP 0.9,-12;
GO TO 410
412 IF ET>WHT THEN BEEP 0.6,-1
2: LET M$="You do not have "+STR
$ ET+" bags!"; GO SUB 1100: PRIM
T: GO TO 410
414 IF ET<POP*AE THEN LET M$="Your
People are hungry!"; GO SUB
1100: LET SF=1
416 IF ET>POP*AE*2 THEN LET M$="Your
People are happy!"; GO SU
B 1100: LET ANG=ANG-1
420 LET WHT=WHT-ET
422 GO SUB 300
424 GO SUB 1000
430 FOR J=22 TO 26 STEP 2
440 FOR K=4 TO 10
450 PRINT AT K,J;CHR$ 147
460 BEEP 0.1,9
480 PRINT AT K,J; "
490 NEXT K
500 PAUSE 50
510 NEXT J
512 LET RTS=INT (RND*WHT/4)
514 LET M$="Rats ate "+STR$ RTS
+" bags of wheat!"; GO SUB 1100
516 LET WHT=WHT-RTS
518 GO SUB 300: GO SUB 1000
520 IF SF=0 THEN GO TO 700
530 LET DD=INT (RND*0.5*(POP*AE
-ET)+1
534 IF DD>=POP THEN LET DD=POP
-1
540 LET M$=STR$ DD+" of your Pe
ople have died of starvation!"
: GO SUB 1100
544 LET POP=POP-DD
550 GO SUB 300: GO SUB 900
560 LET M$="Your People are ang
ry!"; GO SUB 1100

```

```

570 LET ANG=ANG+1
580 IF ANG=3 THEN LET M$="You
have let too many starve. Your
People want a new leader! "; G
O SUB 1100: GO TO 670
590 FOR J=1 TO POP/10
600 IF J>10 THEN GO TO 680
610 PRINT AT 9,J; "
620 PRINT INK 1;AT 9,J;CHR$ 14
5
630 BEEP 0.2,-12
640 PAUSE 5
650 PRINT AT 9,J; "
660 PRINT INK 1;AT 9,J;CHR$ 14
5
670 PAUSE 10
680 NEXT J
690 GO TO 740
700 LET PC=INT (RND*WHT*0.5/AE)
+1
710 LET M$=STR$ PC+" People joi
ned your village!"; GO SUB 1100
720 LET POP=POP+PC
730 GO SUB 300
740 GO SUB 900
750 LET SF=0
760 LET YR=YR+1
770 LET M$="Another year has pa
ssed. "; GO SUB 1100
780 IF YR=11 THEN GO TO 820
800 GO TO 294
820 BEEP 0.3,6
840 BEEP 0.8,12
850 PRINT PAPER 5; INK 9;AT 17
,0;"WELL DONE! YOU HAVE COMPLETE
D YOUR 10 YEARS IN OFFICE."
860 PRINT PAPER 5; INK 9;AT 20
,0;"YOUR SCORE IS ";INT ((POP+WH
T/AE)*10); GO TO 1150
870 FOR J=POP/10 TO 12
872 PRINT INK 1;AT 9,J;CHR$ 14
5
874 PAUSE 5
876 PRINT AT 9,J; "
878 NEXT J
880 PRINT INK 1;AT 10,13;CHR$
```

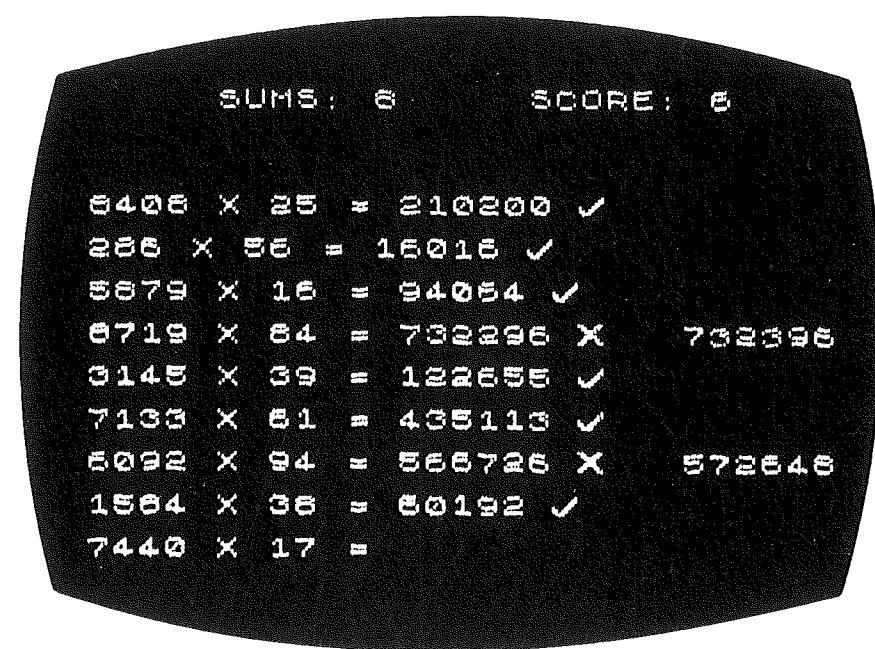
```

145: PAUSE 10
 882 PRINT AT 10,13;" "
 884 PRINT INK 1,AT 11,14,CHR$ 
145: PAUSE 10
 886 PRINT AT 11,14;" "
 888 FOR J=1 TO 5
 890 PRINT INK 2,AT 11,15;"■"
 892 BEEP 0.2,0
 894 PRINT INK '1,AT 11,15,CHR$ 
145
 896 NEXT J
 898 GO TO 1150
 900 LET P$=CHR$ 145
 910 FOR J=9 TO 15
 930 FOR K=1 TO 10
 940 IF P$<K*10+(J-9)*100 THEN
 LET P$=" "
 950 PRINT INK 1,AT J,K,P$ 
 960 NEXT K
 970 NEXT J
 980 RETURN
1000 LET P$=CHR$ 146
1010 FOR J=9 TO 15
1030 FOR K=1 TO 10
1040 IF WHT<K*10+(J-9)*100 THEN
 LET P$=" "
1050 PRINT INK 2,AT J,K+20,P$ 
1060 NEXT K
1070 NEXT J
1080 RETURN
1100 PRINT PAPER 5, INK 9,AT 19
,0,M$ 
1110 PAUSE 500
1120 PRINT AT 19,0;" "
 1130 RETURN
1150 INPUT PAPER 4, INK 9,"PLAY
 AGAIN?",Q$ 
1160 IF Q$(1)="N" OR Q$(1)="n" T
HEN STOP
1170 CLS
1180 GO TO 130

```

2

Multiplication and Division



The first thing you have to do before you begin to play this brain teaser is to put your calculators away in a cupboard and bring out your pencils and paper. This game is a test of your mental ability and agility. No cheating, now, by using any help.

How to play

Your computer will begin by asking you if you wish to play multiplication or division.

Type M or D remembering to use the CAPS key and the ENTER key. You will then be given a simple multiplication question such as 1436×26 .

If you are correct you get a nice little tick and the score will be recorded on the board.

If you are wrong you will receive a large cross against your answer and the correct answer will pop up on the screen.

The program will run for a total of twenty sums and will then give your grand total of correct answers against attempts.

If you wish to change from one type of sum to another you can wait until you have answered your twenty questions then press RUN or you can press caps shift and space together, then RUN. Either will bring you back to the beginning of the program.

Programming hints

You can make the sums easier for younger members of the family by altering lines 200 and 210.

Program

```

10 REM MULTIPLICATION AND
20 REM DIVISION
30 RANDOMIZE
40 LET SM=0
50 LET SC=0
70 FOR N=0 TO 20
80 READ D
90 POKE USR "A"+N,D
100 NEXT N
110 DATA 0,24,0,255,255,0,24,0
120 DATA 0,1,3,6,140,216,112,32
130 DATA 195,102,60,24,60,102,1
95,129
150 INPUT "MULTIPLICATION OR DI
VISION? ",C$
160 IF C$(1)="M" OR C$(1)="m" T
HEN LET S$="X": GO TO 190
165 REM graphics a below
170 IF C$(1)="D" OR C$(1)="d" T
HEN LET S$="a": GO TO 190
180 GO TO 150
185 REM graphics a below
190 IF S$="a" THEN GO TO 260
200 LET R1=INT (RND*9000)+100
210 LET R2=INT (RND*100)+1
220 POKE 23692,255
230 LET CA=R1*R2
240 PRINT AT 21,0;R1;" ",S$;" "
;R2;" = ";

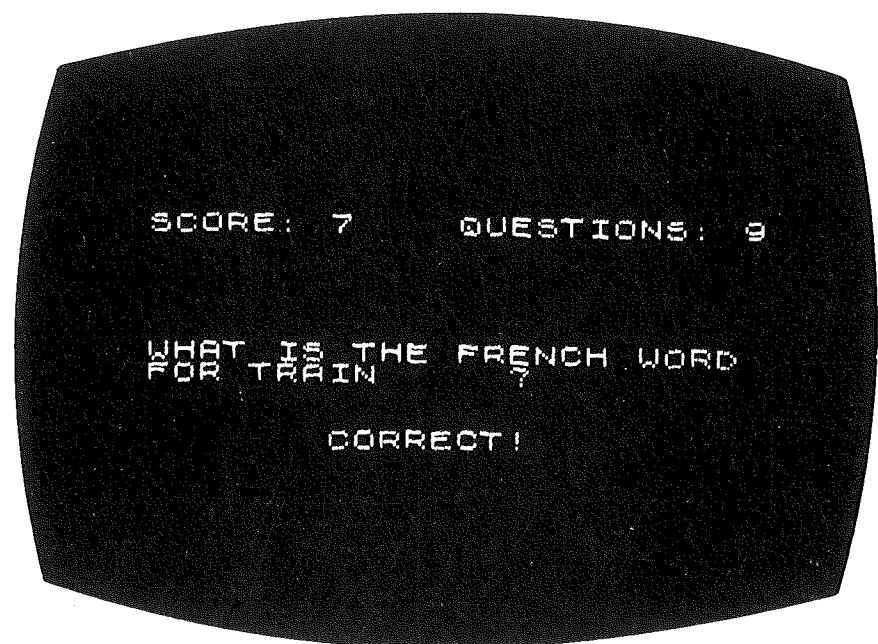
```

```

250 GO TO 310
260 LET R1=INT (RND*100)+1
270 LET R2=INT (RND*100)+1
280 LET CA=R1
290 PRINT AT 21,0;R1*R2;" ",S$;
;" ",R2;" = ";
310 INPUT ANS
320 PRINT ANS;" "
330 LET SM=SM+1
340 IF ANS=CA THEN GO TO 380
350 BEEP 0.9,-12
360 PRINT INK 2;CHR$ 146;" "
;
364 PRINT CA
370 GO TO 410
380 BEEP 0.3,12
390 PRINT INK 2;CHR$ 145
400 LET SC=SC+1
410 PRINT : PRINT
420 IF SM=20 THEN GO TO 460
430 PRINT PAPER 6; INK 9;AT 0,
5;"SUMS: ";SM
440 PRINT PAPER 6; INK 9;AT 0,
17;"SCORE: ";SC
446 PRINT AT 1,0;" "
450 GO TO 190
460 PAUSE 250
470 CLS
480 PRINT PAPER 4; INK 9;AT 8,
3;"YOU HAD ";SC;" CORRECT ANSWER
S"
490 PRINT PAPER 4; INK 9;AT 10
,5;"OUT OF 20 QUESTIONS."
500 STOP

```

3 English/French



Imagine you have gone on holiday with your family to France and your Mum and Dad can't remember the French for an hotel. You just walk up behind them and say 'It's an auberge, Dad'.

They will probably be so surprised that you could knock them down with a pain (that's French for a loaf of bread, but of course you will know that.)

How to play

The computer will concentrate on nouns but you can change the program later to widen your knowledge.

Always remember when answering in French to use le or la before your word or the computer will tell you that you have made an error. Our computers are very correct things you know.

Keep going with your answers, as after twenty correct answers you will get a tasty reward.

Always remember to use your CAPSlock (press CAPS SHIFT and 2 together) to give capital letters. You may be asked questions alternately from French to English then English to French.

Programming hints

Some of the data we have used is very simple so you can change lines 600 onwards to insert your own, or have someone else program, harder examples.

Program

```

10 REM ENGLISH/FRENCH
30 RANDOMIZE
40 LET SC=0
50 LET QN=0
70 DIM E$(20,10)
80 DIM F$(20,15)
100 FOR J=1 TO 20
110 READ E$(J)
120 READ F$(J)
130 NEXT J
140 INPUT "WHAT IS YOUR NAME? "
141 IF LEN(N$)>0 THEN LET N$=N$&
142 IF LEN(N$)>0 THEN LET N$=N$&
143 TO 8)
150 PAUSE 200
154 CLS
160 PRINT PAPER 6; INK 9;AT 0,
5;"SCORE: ",SC
170 PRINT PAPER 6; INK 9;AT 0,
17;"QUESTIONS: ",QN
180 LET R=INT (RND*20)+1
190 IF RND>0.5 THEN GO TO 330
200 PRINT PAPER 4; INK 9;AT 6,
5;"WHAT IS THE FRENCH WORD",
202 PRINT PAPER 4; INK 9;AT 7,
5;"FOR ",F$(R),"?"
210 INPUT A$
220 LET QN=QN+1
230 IF LEN(A$)<15 THEN LET A$=A$&
231 GO TO 230
240 IF A$=F$(R) THEN GO TO 280
250 BEEP 0.7,-12
260 PRINT PAPER 3; INK 9;AT 12
5;"NO ",N$," THE WORD IS"
262 PRINT PAPER 3; INK 9;AT 13

```

```

,5;F$(R)
270 GO TO 150
280 BEEP 0.3,9
290 PRINT PAPER 3; INK 9;AT 10
,12;"CORRECT!"
300 LET SC=SC+1
310 IF SC=20 THEN GO TO 410
320 GO TO 150
330 PRINT PAPER 4; INK 9;AT 6,
5;"WHAT IS THE ENGLISH WORD"
332 PRINT PAPER 4; INK 9;AT 7,
5;"FOR ",F$(R),"?"
340 INPUT A$
350 LET QN=QN+1
360 IF LEN(A$)<10 THEN LET A$=A$&
361 GO TO 360
370 IF A$=E$(R) THEN GO TO 280
380 BEEP 0.7,-12
390 PRINT PAPER 3; INK 9;AT 12
5;"NO ",N$," THE WORD IS"
392 PRINT PAPER 3; INK 9;AT 13
,5;E$(R)
400 GO TO 150
410 PAPER 0
420 BORDER 0
430 CLS
440 PRINT PAPER 5; INK 9;AT 2,
2;"CONGRATULATIONS! YOU HAVE"
450 PRINT PAPER 5; INK 9;AT 3,
2;"ANSWERED 20 QUESTIONS"
460 PRINT PAPER 5; INK 9;AT 4,
2;"CORRECTLY. HERE IS HALF A"
464 PRINT PAPER 5; INK 9;AT 5,
2;"FRENCH ONION TO CHEW! "
470 FOR R=1 TO 36 STEP 3
480 CIRCLE INK 6,125,70,R
490 NEXT R
500 PAUSE 0
510 PAPER 7
520 BORDER 7
530 CLS
540 STOP
600 DATA "TABLE", "LA TABLE"
610 DATA "CHAIR", "LA CHAISE"
620 DATA "DOOR", "LA PORTE"
630 DATA "HOUSE", "LA MAISON"

```

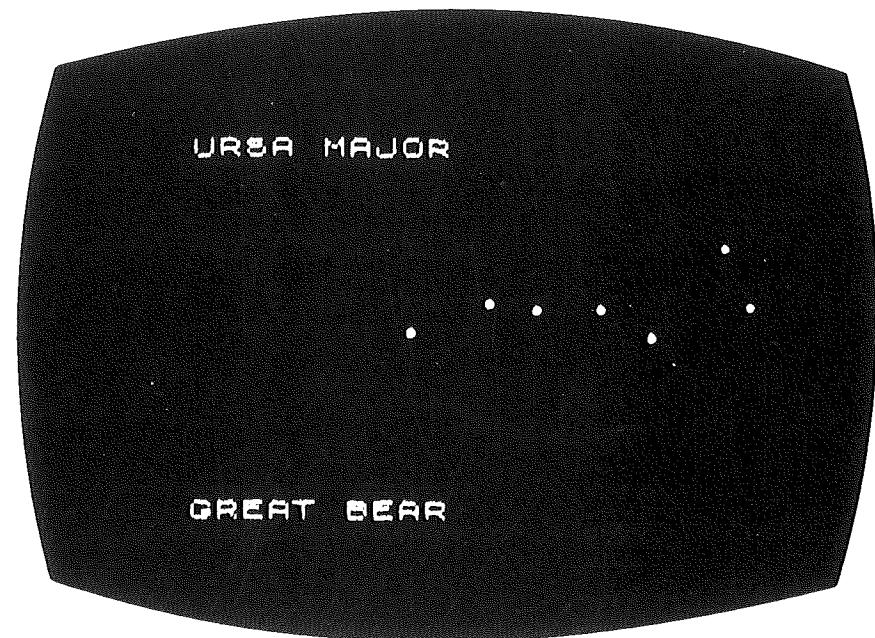
```

640 DATA "DOG", "LE CHIEN"
650 DATA "CAT", "LE CHAT"
660 DATA "GARDEN", "LE JARDIN"
670 DATA "COAT", "LE MANTERU"
680 DATA "HAT", "LE CHAPEAU"
690 DATA "BICYCLE", "LA BICYCLET
TE"
700 DATA "TRAIN", "LE TRAIN"
710 DATA "STATION", "LA GARE"
720 DATA "BREAD", "LE PAIN"
730 DATA "MILK", "LE LAIT"
740 DATA "CUP", "LA TASSE"
750 DATA "APPLE", "LA POMME"
760 DATA "ROAD", "LA RUE"
770 DATA "MAP", "LA CARTE"
780 DATA "SEA", "LA MER"
790 DATA "BOAT", "LE BATEAU"

```

4

Constellations



This is a great game for learning how to make money off your pals by asking them at night if they know which star group is which in the sky.

It is also very handy to know your stars as you never know when you will need to navigate your way out of a crocodile-infested swamp – or drive to London in the dark. The star at the end of the tail of Ursa Minor, or the little bear, is called the pole star and will always be to your North.

How to play

The computer will show you the shape of some of the main star groups to be found in the skies around us and will give you the Latin and the common names for each group.

The screen will then show the stars without any names and will ask you to type in your answer in CAPS and ENTER.

If your guess is wrong the screen will light up and show you both the names. Press the caps shift and space key together to stop the game.

Happy stargazing.

Program

```

10 REM CONSTELLATIONS
20 RANDOMIZE
30 PAPER 0: INK 7: BORDER 0
40 CLS
50 LET A=100: LET B=70
52 LET TT=0
60 FOR C=1 TO 7
62 IF TT=0 THEN GO TO 70
64 LET R=INT (RND*7)+1
66 RESTORE 190+R*10
70 READ X,Y
80 IF X=-99 THEN GO TO 140
90 IF X>199 OR X<-199 THEN LE
T X=X-(200*SGN X): GO TO 120
100 PLOT A+X,B+Y
110 GO TO 70
120 PLOT BRIGHT 1,A+X,B+Y
130 GO TO 70
140 READ M#,N#
142 IF TT=0 THEN GO TO 150
144 INPUT "WHICH IS THIS? ";A#
146 IF A#=M# OR A#=N# THEN BEE
P 0,3,12: PRINT AT 20,4;"THAT IS
CORRECT": PAUSE 150: GO TO 172
148 BEEP 0.8,-12
150 PRINT AT 4,4,M#
160 PRINT AT 21,4,N#
170 PAUSE 1000
172 CLS
180 NEXT C
190 LET TT=1
192 GO TO 60
200 DATA 0,0,24,12,39,9,63,10,7
5,0,305,12,297,33,-99,0,"URSA MA
JOR","GREAT BEAR"
210 DATA 0,0,212,-1,3,19,13,18,
24,33,40,42,257,48,-99,0,"URSA M
INOR","LITTLE BEAR"
220 DATA 0,0,6,-18,18,-18,224,-

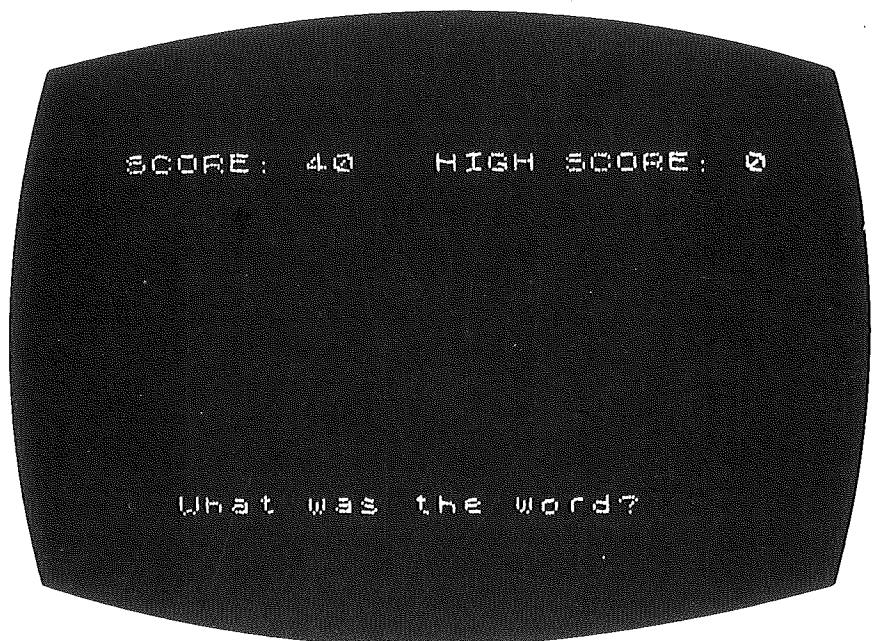
```

```

37,244,-28,-99,0,"CASSIOPEIA",
"
230 DATA 0,-1,5,4,-5,-5,-6,-35,
12,30,226,-30,-220,33,-99,0,"ORI
ON","THE HUNTER"
240 DATA 0,0,7,-2,14,-1,220,2,2
5,10,20,19,-99,0,"CORONA BOREALI
S","NORTHERN CROWN"
250 DATA 0,0,21,-9,39,-12,38,-1
9,-99,0,"SAGITTA","THE ARROW"
260 DATA 0,0,25,12,234,55,247,3
5,90,-9,72,8,66,15,74,52,-99,0,"
CYGNUS","THE SWAN"

```

5 Spelling Test



Well, you must have expected to find a spelling test somewhere in this section and here it is.

How to play

Your computer will put a word on the screen for a few seconds and then blank the screen and ask you to spell the same word correctly.

You must use lower case (small letters) for this game. Remember you can change from small letters to capitals and vice versa by pressing CAPS SHIFT and 2 together.

Programming hints

If you think the time that you see the word on the screen is too long or too short you can alter the pause in line 170. If you feel that the words are too easy for a

bright young thing like you then ask your parents to type in some more difficult words from line 400 onward. If you add more words, change the 50 in lines 70, 80, and 120 to match the total number of words.

The computer will ask its questions at random so you shouldn't really know what is coming up next.

Program

```

10 REM SPELLING TEST
30 RANDOMIZE
40 LET SC=0
60 LET HS=0
70 DIM WS(50,10)
80 FOR J=1 TO 50
90 READ WS(J)
100 NEXT J
110 FOR N=1 TO 20
120 LET R=INT (RND*50)+1
130 CLS
140 PRINT AT 0,4;"SCORE: ";SC
150 PRINT AT 0,16;"HIGH SCORE:
",HS
160 PRINT AT 8,9;WS(R)
170 PAUSE 70
180 PRINT AT 8,9;" "
190 PRINT AT 16,6;"What was the
word?"
200 INPUT A$
202 IF LEN A$<10 THEN LET A$=A$+
": GO TO 202
210 IF A$=WS(R) THEN GO TO 270
220 BEEP 1,-12
230 PRINT AT 8,5;"The word was
",WS(R)
244 PRINT AT 16,3;""
250 PAUSE 400
260 GO TO 300
270 BEEP 0,3,12
280 LET SC=SC+10
300 NEXT N
310 IF SC>HS THEN LET HS=SC
312 CLS
314 PRINT AT 8,3;"Your final score
was ";SC
316 IF SC=HS THEN PRINT AT 10,

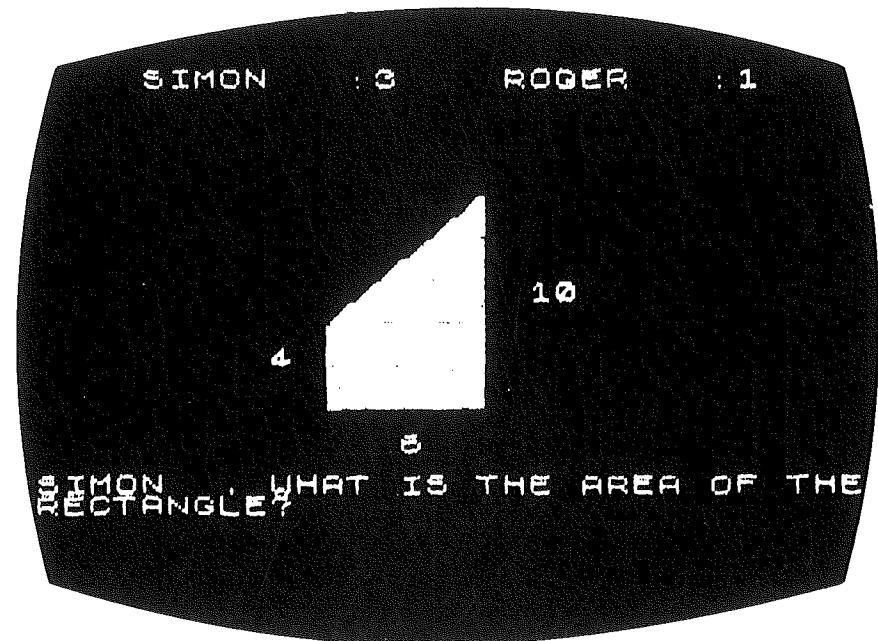
```

```

0;"This is the highest score to
ay."
320 INPUT "Play again?";Q$
330 IF Q$(1)="n" OR Q$(1)="N" T
HEN GO TO 380
340 LET SC=0
360 GO TO 110
380 STOP
400 DATA "quiz","message","Pave
ment"
410 DATA "bicycle","special","b
eneath"
420 DATA "mountain","listen","s
chool"
430 DATA "tomorrow","business",
"address"
440 DATA "parallel","height","l
ength"
450 DATA "ceiling","expert","ke
ttle"
460 DATA "colonel","surprise","
forecast"
470 DATA "attach","rhubarb","me
rinque"
480 DATA "daffodil","knowledge"
,"yacht"
490 DATA "tongue","miniature",
"dinghy"
500 DATA "amateur","punctual",
"illogical"
510 DATA "giraffe","parsley","t
riangle"
520 DATA "legible","mosaic","di
sciple"
530 DATA "amend","guitar","bell
eve"
540 DATA "station","presence",
"saviour"
550 DATA "alcohol","cabaret","s
wallowable"
560 DATA "aquatic","pneumatic"

```

6 Areas



This game is a bit of a brain teaser and might drive you mad before you start to get it right. To begin with you might even need a pencil and some paper but really that is a bit of a cheat.

The object is to work out the total area of a rectangle and a triangle – together. Your parents will be highly impressed when you can tell them how much carpet they need for that funny shaped room upstairs.

How to play

You will be asked 'One or two players?' Type in 1 or 2 and press ENTER.

You will then be asked for the player's, or players', names. ENTER as before.

The computer will then show you a green rectangle with a red triangle on top. You will be given the length of each of the sides and asked to work out the total area.

If you do not get the answer correct first time the computer will then ask you to give the area of the rectangle first. If you answer this part correctly you will then be shown the triangle again and asked for its area. If either part of your answer is wrong the computer will give you the correct total area sum.

If you get the answer right first time, clever clogs, you will be given two points but if you answer in two halves you will get only one point for your correct answers.

To help you bend the rules the formula for area is

$$CXB + \frac{1}{2}B(X-A-C)$$

Program

```

10 REM AREAS
20 LET CP=40
22 LET LP=90
30 DIM S(2)
40 DIM P$(2,7)
50 RANDOMIZE
60 INPUT "1 OR 2 PLAYERS? ",N
70 IF N<1 OR N>2 THEN GO TO 6
0
80 PRINT AT 10,2;"WHAT ARE THE
PLAYERS NAMES?"
90 FOR J=1 TO N
100 PRINT AT 12,4;"PLAYER ",J
110 INPUT P$(J)
120 NEXT J
122 CLS
130 FOR L=1 TO N
140 LET A=INT (RND*7)+4
150 LET B=INT (RND*6)+3
160 LET C=INT (RND*5)+3
162 IF C>=A THEN GO TO 140
172 LET SA=A*B
174 LET SB=B*B
176 LET SC=C*B
180 FOR J=CP TO CP-1+SC
190 PLOT LP,J
200 DRAW INK 4,SB,0
210 NEXT J
220 PLOT LP,CP+SC
222 DRAW INK 2,SB,SA-SC
224 DRAW INK 2,0,-(SA-SC)
226 FOR J=CP+SC TO CP-1+SA
228 FOR K=LP-1+SB TO LP STEP -1

```

```

230 IF POINT (K,J)=1 THEN LET
K=60: GO TO 234
232 PLOT INK 2,K,J
234 NEXT K
236 NEXT J
240 PRINT AT 16-INT A/2,13+B,A
250 PRINT AT 18,INT 11+B/2,B
260 PRINT AT INT 16-C/2,9,C
270 PRINT AT 20,0;P$(L);", WHAT
IS THE TOTAL AREA?
"
280 INPUT ANS
290 IF ANS=C*B+B/2*(A-C) THEN
GO TO 550
310 BEEP 0.8,-8
320 PRINT AT 20,0;P$(L);", WHAT
IS THE AREA OF THERECTANGLE?
"
330 INPUT RC
340 IF RC=C*B THEN GO TO 370
350 BEEP 0.8,-8
360 PRINT AT 20,0;"THE AREA OF
THE RECTANGLE IS ",(C*B);"
"
362 PAUSE 200
364 GO TO 500
370 PRINT AT 20,0;P$(L);", WHAT
IS THE AREA OF THETRIANGLE?
"
380 INPUT TG
390 IF TG=B/2*(A-C) THEN GO TO
420
400 BEEP 0.8,-8
410 PRINT AT 20,0;"THE AREA OF
THE TRIANGLE IS ",B/2*(A-C);"
"
412 PAUSE 200
414 GO TO 500
420 PRINT AT 20,0;P$(L);", WHAT
IS THE TOTAL AREA?
"
430 INPUT ANS
440 IF ANS=C*B+B/2*(A-C) THEN
GO TO 470
450 BEEP 0.8,-8
460 GO TO 500
470 BEEP 0.3,8

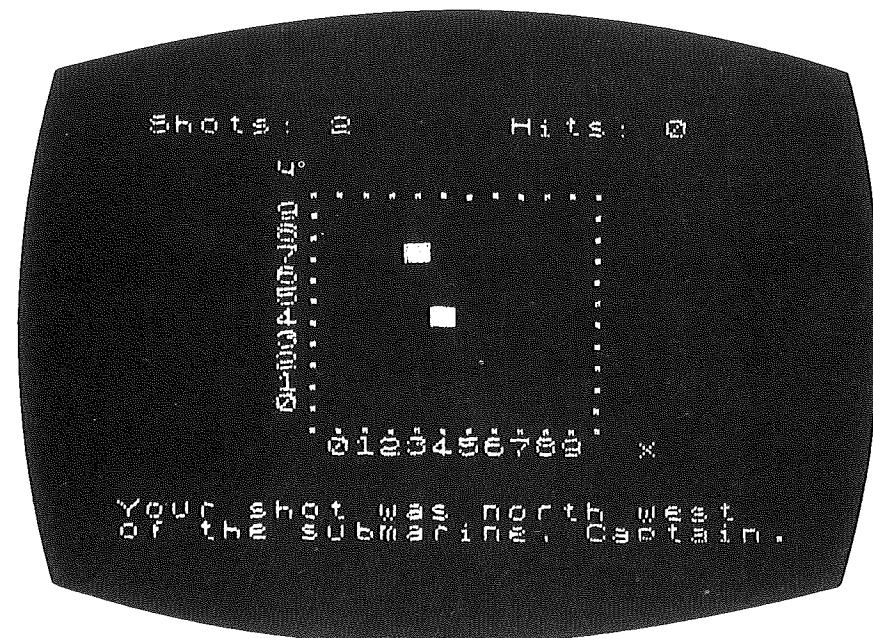
```

```

400 LET S(L)=S(L)+1
490 GO TO 562
500 PRINT AT 20,0;"THE TOTAL AREA IS ",C*B+B/2*(A-C),"
"
502 PAUSE 200
506 CLS
510 GO TO 570
550 BEEP 0.3,8
560 LET S(L)=S(L)+2
562 CLS
570 PRINT AT 1,4;P(1);"-";S(1)
">;"-;P(2);"-";S(2)
580 NEXT L
600 GO TO 130

```

7

Submarine

You are a destroyer captain alone in a hostile sea surrounded by a pack of submarines which are travelling secretly to a rendezvous. The submarines cannot break radio silence or send for help and must not attack you for fear of giving their position away but you can sink as many of them as you can – with as few depth charges as possible.

How to play

On the screen will be shown a board divided into 100 squares. The submarine is hiding in one of those squares. The bottom (horizontal) line is called X and the upright (perpendicular) line is called Y.

Each line of boxes goes from 0 to 9 and you have to give the box numbers to the computer when it asks for your entry. You will be asked to type in a number for the X and Y lines. If you think that the submarine is in a box 8 across and 5 high then ENTER 8 and 5 when the X and Y positions are asked for.

Remember you must press ENTER after each number.

After the second number is entered you will hear the 'crump' of an exploding depth charge. If you make a direct hit first time you will hear a 'whooping' sound and the screen will show you how many tries you took to sink the submarine.

If however you miss, the computer will tell you if your shot was North, South, East or West of the target and you must then plan your next shot.

As soon as the submarine is sunk your computer will search and detect another target.

Special Note. Expert captains should be able to detect and sink the enemy within four moves.

Programming notes

The square in lines 245 and 435 is a graphics symbol – hold down CAPS SHIFT and press 9, then 8, then 9 again.

Program

```

10 REM SUBMARINE
30 INK 7: PAPER 0: BORDER 0
32 CLS
40 RANDOMIZE
50 PRINT INK 2:AT 0,3;"Shots:
0"
60 PRINT INK 2:AT 0,17;"Hits:
0"
70 PRINT AT 2,8;"y"
80 PRINT AT 15,22;"x"
90 FOR J=10 TO 19
100 PRINT INK 5:AT 3,J;"."
110 PRINT INK 5:AT 14,J;"."
120 PRINT INK 5:AT 15,J;J-10
130 NEXT J
140 FOR J=3 TO 14
150 PRINT INK 5:AT J,9;"."
160 PRINT INK 5:AT J,20;"."
170 NEXT J
180 FOR J=4 TO 13
190 PRINT INK 5:AT J,6;13-J
200 LET sh=0: LET ht=0
205 LET ax=0: LET ay=0
210 NEXT J
220 LET sy=INT (RND*10)
230 LET sx=INT (RND*10)

```

```

240 INPUT "Your shot (x,y)? ";a
x,ay
242 IF ax<0 OR ax>9 OR ay<0 OR
ay>9 THEN GO TO 240
245 PRINT INK 2:AT 13-ay,10+ax
;"■"
250 FOR J=1 TO 10
260 BEEP 0,1,9-J
270 NEXT J
290 PRINT AT 18,2;""
300 PRINT AT 18,2;
310 IF ax=sx AND ay=sy THEN GO
TO 420
320 PRINT "Your shot was ";
330 IF ay<sy THEN PRINT "south
"
340 IF ay>sy THEN PRINT "north
"
350 IF ax<sx THEN PRINT "west
"
360 IF ax>sx THEN PRINT "east
"
370 PRINT AT 19,2;"of the subma
rine, Captain."
380 LET sh=sh+1
390 PRINT INK 2:AT 0,10;sh
400 PAUSE 200
410 GO TO 235
420 FOR J=1 TO 20
430 BEEP 0,1,9
435 PRINT INK 2:AT 13-ay,10+ax
;"■"
440 NEXT J
450 PRINT AT 18,2;"A direct hit
"
452 FOR J=4 TO 13
454 PRINT AT J,10;""
456 NEXT J
460 LET ht=ht+1
470 PRINT INK 2:AT 0,23;ht
472 IF ht=10 THEN GO TO 540
480 PAUSE 300
490 PRINT AT 18,2;"Another subm
arine has been detected!"

```

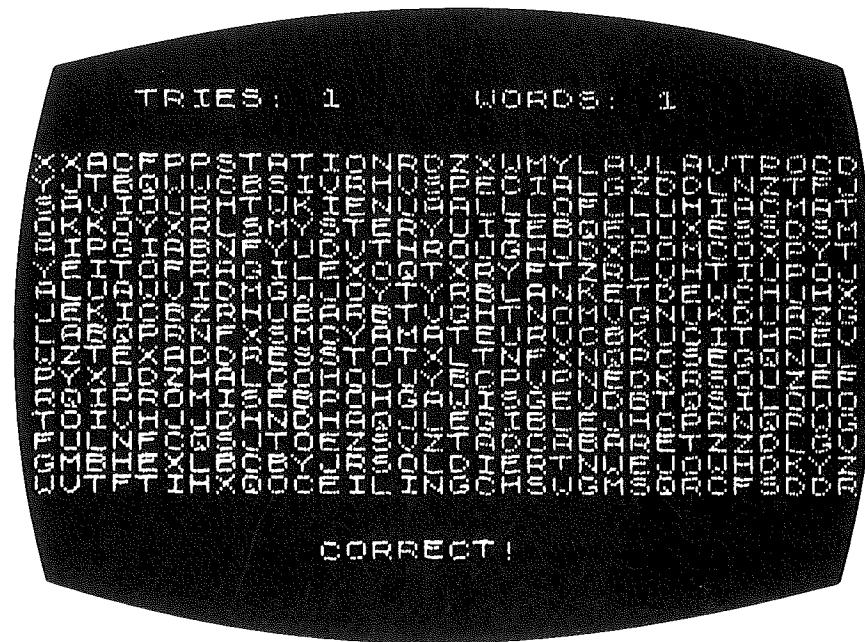
```

510 GO TO 220
540 FOR J=1 TO 10
550 BEEP 0.5,J
560 NEXT J
562 CLS
570 PRINT AT 8,8;"CONGRATULATIO
NS!"
580 PRINT AT 10,6;"TEN SUBMARIN
ES SUNK!"
590 PRINT AT 13,9;"YOU HAVE BEE
N"
600 PRINT AT 14,6;"PROMOTED TO
ADMIRAL!"

```

8

Word Search



This is a game to see how sharp your eyesight is and how quick you are at spelling words which are hidden on the screen.

The computer will put up a selection of letters all over the screen, and hidden amongst this alphabet spaghetti will be certain words which you will have to spot and spell out to your computer.

How to play

When you find a word on the screen you type it in, remembering to use CAPITAL letters. You then press the ENTER key and, if you have correctly identified and spelled out the word, it will be picked out, in green, on your screen.

The score board will show your number of tries and the number of words you have correctly spotted.

When you cannot find any more words you simply type in the word QUIT.

Your computer will then blank out the words you have found and show, in red, all the words you have missed.

Your score will be shown on the screen.

Programming hints

If the selection of words is too simple or difficult you can change the data in lines 500 to 530. Make sure that all the words you use have exactly 7 letters.

Program

```

10 REM WORD SEARCH
30 RANDOMIZE
40 DIM m(20)
47 DIM b(17)
50 DIM w(20,7)
52 DIM c(20,7)
55 LET sc=0: LET tr=0
60 FOR j=1 TO 20
70 READ w(j)
80 NEXT j
90 PRINT AT 3,0
100 FOR k=1 TO 16
110 LET r1=INT (RND*20)+1
115 IF m(r1)=1 THEN GO TO 110
117 LET m(r1)=1
119 LET c(k)=w(r1)
120 LET r2=INT (RND*20)+1
125 LET b(k)=r2-1
130 FOR l=1 TO 32
140 IF l=r2 THEN PRINT c(k))
LET l=l+7
150 LET lt=INT (RND*26)+65
160 PRINT CHR$(lt)
170 NEXT l
180 PRINT
190 NEXT k
200 INPUT "YOUR WORD? ",a
202 IF a="QUIT" THEN GO TO 40
204 IF a>="a" AND a<="z" THEN
PRINT AT 20,4;"USE CAPITALS PLEASE": GO TO 200
205 LET tr=tr+1
207 PRINT AT 21,11,""
210 FOR j=1 TO 16

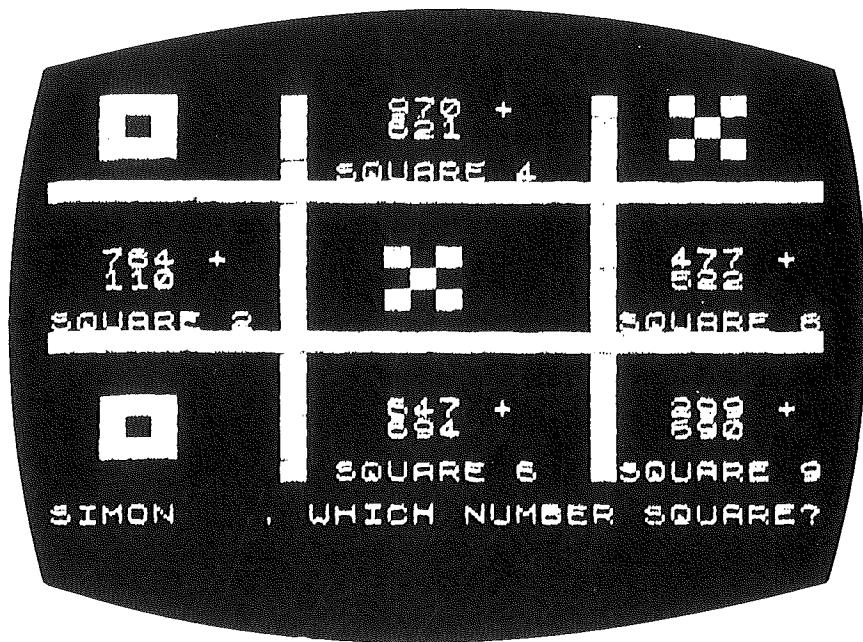
```

```

220 IF a=c(j) THEN GO TO 270
230 NEXT j
240 BEEP 0.3,-4
250 PRINT PAPER 3: INK 9;AT 21
,12;"WRONG!"
260 GO TO 300
270 BEEP 0.2,4
275 PRINT PAPER 4: INK 9;AT j+
2,b(j);c(j)
280 PRINT PAPER 3: INK 9;AT 21
,11;"CORRECT!"
285 LET c(j)=""
290 LET sc=sc+1
300 PRINT PAPER 2: INK 9;AT 0,
4;"TRIES: ",tr
310 PRINT PAPER 2: INK 9;AT 0,
17;"WORDS: ",sc
320 IF sc>16 THEN GO TO 200
330 STOP
400 FOR j=1 TO 16
410 PRINT PAPER 2: INK 9;AT j+
2,b(j);c(j)
420 NEXT j
430 GO TO 330
500 DATA "PROMISE", "SPECIAL", "BLANKET", "FRAGILE", "THROUGH"
510 DATA "ADDRESS", "CEILING", "AMATEUR", "MYSTERY", "LEGIBLE"
520 DATA "BELIEVE", "STATION", "RHUBARB", "DISSECT", "SAVIOUR"
530 DATA "ALCOHOL", "CUSHION", "OLDIER", "CABARET", "AQUATIC"

```

Noughts and Crosses



No, it's not that same old boring game that you play when it's raining outside and there is nothing else to do.

With our noughts and crosses you have to solve a problem **before** you can make your mark on your computer's board. You will have to decide if you can answer the question before you choose your box.

How to play

The computer will ask for the players' names which you type in and enter by pressing the ENTER key.

The computer will then present you with a board and in each of the nine squares you will find a maths sum to solve.

First choose the box you want and ENTER the number. Now you can type in your answer and see if you get your nought or cross for the correct answer. Your opponent must follow the same steps.

If either player gives a wrong answer the computer moves to the opposing player's turn.

The game progresses until someone gets a winning line at which point you can press caps shift and space together, then RUN to play again.

If no one can complete a winning line then the game can proceed until all the boxes are completed and the player with the highest number of noughts or crosses will be the winner.

Programming note

You can make the sums easier by reducing the numbers in line 200 or more difficult by typing in a larger number. The squares in lines 120, 130, 160, 170, 500, 510, 520, 560, 580 and 600 are all graphics squares. Press CAPS SHIFT and 9 together for graphics mode then CAPS SHIFT and 8 together for each square.

Program

```

10 REM NOUGHTS AND CROSSES
30 RANDOMIZE
40 DIM NC(18)
42 DIM P$(2,8)
44 DIM H(9)
50 PRINT AT 10,2;"WHAT ARE THE
PLAYERS NAMES?"
60 INPUT "PLAYER ONE: ",P$(1)
70 INPUT "PLAYER TWO: ",P$(2)
80 CLS
110 FOR J=2 TO 19
120 PRINT INK 5;AT J,9;"■"
130 PRINT INK 5;AT J,21;"■"
140 NEXT J
150 FOR J=0 TO 29
160 PRINT INK 5;AT 6,J;"■"
170 PRINT INK 5;AT 13,J;"■"
180 NEXT J
190 FOR J=1 TO 18
200 LET NC(J)=INT (RND*900)+100
210 NEXT J
220 LET C=1
230 FOR J=2 TO 24 STEP 11
240 FOR K=2 TO 16 STEP 7
250 PRINT AT K,J;NC
260 LET C=C+1
270 PRINT AT K+1,J;NC
280 LET C=C+1

```

```

290 PRINT AT K,J+4;"+"
300 PRINT AT K+3,J-2;"SQUARE ";
INT (C/2)
310 NEXT K
320 NEXT J
324 FOR N=1 TO 2
330 PRINT AT 21,0;P$(N);", WHICH
NUMBER SQUARE?"
340 INPUT SQ
341 IF SQ<1 OR SQ>9 THEN GO TO
330
342 PRINT AT 21,0;"SQUARE ";SQ;
"
350 INPUT "WHAT IS THE ANSWER?
",ANS
360 IF ANS=N*(SQ*2)+N*(SQ*2-1) TH
EN GO TO 420
370 BEEP 1,-6
400 GO TO 620
420 LET H(SQ)=1
422 BEEP 0,4,8
430 IF SQ>0 AND SQ<4 THEN LET
W=2: LET V=SQ*7-5
440 IF SQ>3 AND SQ<7 THEN LET
W=13: LET V=(SQ-3)*7-5
450 IF SQ>6 AND SQ<10 THEN LET
W=24: LET V=(SQ-6)*7-5
460 FOR J=0 TO 3
470 PRINT AT J+V,W-2;""
480 NEXT J
490 IF N=2 THEN GO TO 560
500 PRINT INK 2;AT V+0,W;"■■"
510 PRINT INK 2;AT V+1,W;"■"
520 PRINT INK 2;AT V+2,W;"■■"
550 GO TO 620
560 PRINT INK 4;AT V+0,W;"■■■■"
580 PRINT INK 4;AT V+1,W;"■■"
600 PRINT INK 4;AT V+2,W;"■■■■"
620 NEXT N
630 FOR J=1 TO 9
640 IF H(J)=0 THEN GO TO 324
650 NEXT J
660 STOP

```

Chapter Fourteen

Games for Experienced Users

Don't be too put off by the title of this chapter. If you've programmed the games in Chapter 13 without any trouble, then you shouldn't have much more with these games. However, most of these listings are rather long, so you will need to take extra care about entering them absolutely correctly; before you start you might like to take another quick look at the introduction to this section of the book.

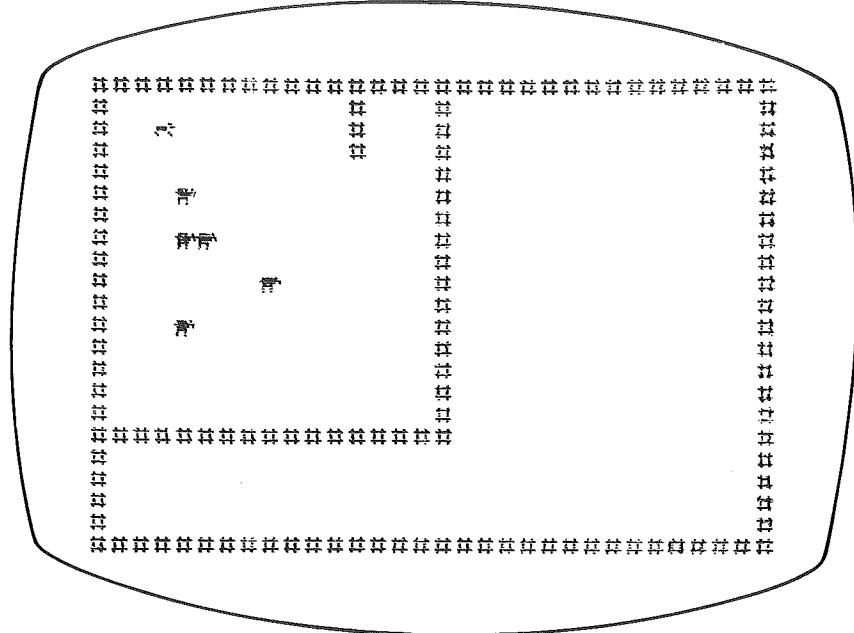
Listing conventions

These programs have not been reproduced using the ZX Printer (like those, for instance, in Part 1) because we wanted to lay them out in a more easy-to-read form with lines split, where necessary, at points that preserved their syntax and blank lines between subroutines. The listings also show how you should type the program in, rather than how they will look once they are typed. For example, the keywords GOSUB and GOTO appear as single words, just as they do on the Spectrum keyboard, but you won't need to leave spaces after these words when you're typing – the Spectrum does that for you. In fact the only place where you'll have to use the SPACE key is when you want to have a space within a PRINT statement. All characters that need to be entered in graphics mode have been enclosed in square brackets, so if you need to type the letter 'a' in graphics mode you'll see [a] in the program listing. Sometimes you will need to hold the CAPS SHIFT down while you're in graphics mode to produce graphics characters (i.e. the characters on the number keys). Whenever you need to use CAPS SHIFT in graphics mode the symbol ^ will be printed before the number in question. For instance, [^ 1] indicates that you should be in graphics mode, then press the 1 key with CAPS SHIFT held down. Each program is accompanied by a set of 'typing tips', so you may see this explanation repeated a couple of times.

The only problem with using a different printer is that the zeros appear without an oblique stroke through them, but this shouldn't cause any confusion. If you ever need to type an upper case letter 'O', or 'o' is used as a variable name, you'll be alerted in the typing tips section. To avoid any confusion between the lower case letter 'l' and the number '1', every 'l' used as a variable name is shown in capitals. However, you don't need to copy this convention when you're typing in programs yourself.

And now – on with the games!

Sheepdog Trials



If you've ever watched a shepherd and his dog coax a flock of sheep into a pen you're bound to agree that it's quite an astounding feat. The experienced shepherd makes it all look so effortless as he shouts and whistles commands to his dog who obediently stands his ground or edges up a few paces or runs around the back of the flock to head off a straggler.

This game is an extremely realistic simulation. In fact it's so true-to-life that the only person we know who's scored a 'Super Shepherd' rating was a real sheep farmer! Five fluffy white sheep and a black dog appear in a green field surrounded by a picket fence – it creates just the right country atmosphere.

How to play

The object of the game is to herd all five sheep into the pen at the top left-hand side of their field in the minimum number of moves. To do this you have to control the dog using the arrow keys (keys 5, 6, 7 and 8). If the dog approaches too close to the sheep they will scatter. (They also scatter randomly during the course of the game just to complicate matters.) In normal play neither the dog nor the sheep are

allowed to cross any fences, although when they scatter the sheep may jump out of the pen. There will always be a total of five sheep but if they crowd very close together they will appear to merge into one another.

Once you've played this game a few times you'll realise that some strategies for controlling the sheep will work better than others. Beginners tend to waste moves trying to manoeuvre the dog around the back of the flock. However, to achieve the title of 'Super Shepherd' or 'Good dog' you'll need to make every move count.

Typing tips

Wherever you see a character within square brackets remember that this indicates that you must use graphics mode. In this program you will find [s] in lines 1040 and 7350 and [d] in lines 1130 and 1660. The hash character used to print the fence in subroutine 800 is produced by typing the 3 key with SYMBOL SHIFT held down. The only other printing feature to look out for is the single space enclosed in double quotes in lines 1610 and 7300. You may be surprised by the inclusion of '+0' in lines 500 and 600. This is actually an unnecessary embellishment to the POKE USR statement and has only been included in order to make it easier for you to pick out the shapes of the user-defined graphics. For the same reason, you may find it easier to type these lines as they appear but if you prefer you can omit the '+0'.

Subroutine structure

| | |
|------|--|
| 500 | Defines sheep graphics character |
| 600 | Defines dog graphics character |
| 700 | Initialises arrays |
| 800 | Prints fences |
| 1000 | Prints sheep |
| 1100 | Prints dog |
| 1500 | Moves dog |
| 1700 | Moves sheep and checks for end of game |
| 7000 | Move logic for sheep |
| 7300 | Prints sheep |
| 8000 | Scatters sheep |
| 9000 | Prints messages at end of game and offers another go |

Programming details

Lines 7100 and 7150 check to see if the dog has approached too close to the sheep. If he has (or if the random number generated is less than .01 - a one-in-a-hundred chance occurrence) then the sheep scatter according to the equations in 8050 and 8070. Notice the use of ATTRibute statements in lines 1620 and 1650. They are used to make sure that the dog does not move into any of the picket fences. Similar

statements are used in lines 7170 and 7175 where they ensure that the sheep do not move on to the dog or on to the picket fence.

Scope for improvement

If you get really proficient at this game you can try to make it more difficult. You might increase the chance of the sheep scattering at random, by altering the value of the cut-off point for the random number in line 7100 or you could add some obstacles such as a pond or a river that the sheep have to avoid or cross. Another suggestion is to modify the game to employ a time criterion, using the Spectrum's clock, instead of counting the number of moves needed.

Program

```

100 REM Sheepdog trial

500 POKE USR "'s"+0,BIN 00000000
510 POKE USR "'s"+1,BIN 00000000
520 POKE USR "'s"+2,BIN 01111010
530 POKE USR "'s"+3,BIN 11111111
540 POKE USR "'s"+4,BIN 01111101
550 POKE USR "'s"+5,BIN 01111000
560 POKE USR "'s"+6,BIN 01001000
570 POKE USR "'s"+7,BIN 01001000

600 POKE USR "'d"+0,BIN 00000000
610 POKE USR "'d"+1,BIN 00000000
620 POKE USR "'d"+2,BIN 00000110
630 POKE USR "'d"+3,BIN 01111011
640 POKE USR "'d"+4,BIN 01111000
650 POKE USR "'d"+5,BIN 10000100
660 POKE USR "'d"+6,BIN 01000010
670 POKE USR "'d"+7,BIN 00000000

700 DIM s (5)
710 DIM x (5)
720 LET m=0

800 PAPER 4:INK 4
810 CLS
820 FOR x=0 TO 15
830 PRINT AT 16,x; INK 0;"#"
840 NEXT x
850 FOR y=0 TO 16
860 PRINT AT y,16; INK 0;"#"
870 NEXT y

```

```

900 FOR y=0 TO 20
910 PRINT AT y,0; INK 0;"#"; AT y,31;"#"
920 NEXT y
930 FOR x=0 TO 31
940 PRINT AT 0,x; INK 0;"#"; AT 21,x;"#"
950 NEXT x
960 FOR y=1 TO 3
970 PRINT AT y,12; INK 0;"#"
980 NEXT y

```

```

1000 REM print sheep
1010 FOR s=1 TO 5
1020 LET y(s)=5+INT (RND*10)
1030 LET x(s)=4+INT (RND*6)
1040 PRINT AT y(s),x(s); INK 7;"Es]"
1050 NEXT s

```

```

1100 REM print dog
1110 LET yd=1+INT (RND*3)
1120 LET xd=1+INT (RND*3)
1130 PRINT AT yd,xd; INK 0;"Ed]"

```

```

1500 REM move dog
1590 LET d$=INKEY$
1600 IF d$="" THEN GOTO 1590
1610 PRINT AT yd,xd;""
1620 IF d$="5" AND ATTR (yd,xd+1)>>32
  THEN LET xd=xd+1
1630 IF d$="8" AND ATTR (yd,xd+1)>>32
  THEN LET xd=xd+1
1640 IF d$="6" AND ATTR (yd+1,xd)>>32
  THEN LET yd=yd+1
1650 IF d$="7" AND ATTR (yd-1,xd)>>32
  THEN LET yd=yd-1
1660 PRINT AT yd,xd; INK 0;"Ed]"
1670 LET m=m+1
1680 PRINT INK 0; AT 10,20;"Move ";m

```

```

1700 GOSUB 7000
1820 IF f=0 THEN GOTO 9000
1850 GOTO 1500

```

```

7000 REM move sheep
7010 LET f=0
7050 FOR s=1 TO 5
7070 LET y=y(s);LET x=x(s)

```

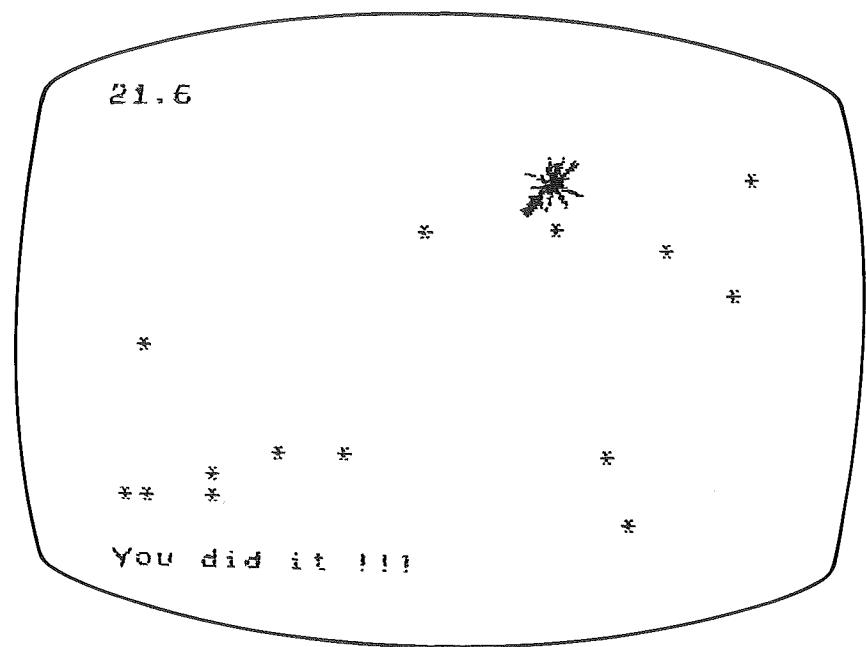
```

7080 IF (ABS (x(s)-xd)<2 AND
  ABS (y(s)-yd)<2) OR (RND<.01) THEN
  GOSUB 8000
7100 IF ABS(x(s)-xd)>2+INT (RND*2)
  OR ABS(y(s)-yd)>2+INT (RND*2) THEN
  GOTO 7125
7150 LET x(s)=x(s)+SGN(x(s)-xd)
7160 LET y(s)=y(s)+SGN(y(s)-yd)
7170 IF ATTR (y(s),x(s))=39 THEN
  GOTO 7150
7175 IF ATTR (y(s),x(s))=32 THEN
  LET x(s)=x;LET y(s)=y+1
7180 IF x(s)<1 THEN LET x(s)=1
7190 IF x(s)>14 THEN LET x(s)=14
7260 IF y(s)<1 THEN LET y(s)=1
7270 IF y(s)>14 THEN LET y(s)=14
7300 PRINT AT y,x; INK 4;""
7350 PRINT AT y(s),x(s); INK 7;"Es]"
7360 IF x(s)>12 AND (y(s)=1 OR y(s)=2
  OR y(s)=3) THEN GOTO 7400
7370 LET f=1
7400 NEXT s
7500 RETURN
8000 REM scatter sheep
8050 LET x(s)=x(s)+((SGN (RND-.5))*(
  2+INT (RND*2)))
8070 LET y(s)=y(s)+((SGN (RND-.5))*(
  2+INT (RND*2)))
8100 RETURN
9000 PRINT AT 18,2;
9005 IF m<40 THEN PRINT INK 0;
  "Super Shepherd!!";GOTO 9080
9010 IF m<60 THEN PRINT INK 0;
  "Good Dog!!";GOTO 9080
9020 IF m<90 THEN PRINT INK 0;
  "Keep practising";GOTO 9080
9025 IF m<120 THEN PRINT INK 0;
  "Better luck next time";GOTO 9080
9030 PRINT INK 0;"Hand in your crook !!"
9080 PRINT AT 20,2; INK 0;"You took ";m;
  " moves";
9090 INPUT "Another game (y/n) ";a$
9100 IF a$(1)="y" THEN RUN
9110 PAPER 7;INK 0
9120 STOP

```

2

Laser Attack



This is a very exciting game that uses some rather unusual graphics techniques to good effect. The screen is treated as if it were a spherical universe. So, if you go off the top of the screen you reappear at the bottom and if you go off at the right you reappear at the left. This game is a race against the clock. You have a hundred seconds in which to annihilate the enemy ship with your infallible laser weapon – the chase is on.

How to play

At the beginning of this game you have to select a difficulty factor. This governs the unpredictability of the enemy ship's course and the number of stars that appear. The stars act as obstacles in this game. If you hit one you will be deflected at random, so the fewer there are the easier it is to steer your course. Your ship moves continuously. It is shaped like an arrow-head and can point in any of eight directions. Every time you press any key it turns clockwise through 45 degrees. The enemy is a revolving cartwheel-shaped disc that meanders through space. To fire your laser press the up arrow key. Your weapon will fire in a straight line from

the point of your arrow. If you hit the enemy ship it will disintegrate with appropriate sound and visual effects. The time taken is constantly displayed at the top left of the screen and when it reaches 100 your time is up.

Typing tips

Look out as usual for graphics characters enclosed in square brackets. Remember to type the letters so indicated in graphics mode and without the brackets. Also, as elsewhere in this book, where 'I' is used as a variable name, e.g. in line 2000, it has been printed out as a capital just to avoid confusion with the number '1'.

Subroutine structure

```

20 Main play loop
1000 Fires or rotates direction of arrow
2000 Laser zap and detect hit logic
3000 Moves arrow
4000 Moves target
7000 Title frame
7500 Gets and prints time
7900 Prints stars
8000 Initialises variables and defines arrow graphics
8600 Defines cartwheel graphics
8810 Zeroes time
9000 End of game

```

Programming details

This is a complicated program and one that illustrates a number of novel programming methods. Notice, for example, the way the revolving cartwheel is produced by using two user-defined graphics characters, one a version of the other rotated through 90 degrees, which are printed alternately. Also notice that eight versions of the arrow graphic are used in order to allow it to be moved in any of eight different directions – also an interesting technique. The way the direction of movement and velocity is set using the arrays 'w' and 'v' is also worth attention.

Scope for improvement

If you feel very ambitious you could make this game even more exciting by enabling the enemy ship to fire at random – so that you have to dodge its fire at the same time as pursuing it.

Program

```

10 REM Laser attack
20 GOSUB 7000
30 GOSUB 8000
40 GOSUB 7500
50 IF INT t>1000 THEN GOTO 9000
60 GOSUB 1000
70 GOSUB 3000
80 IF h=1 THEN GOTO 9000
90 GOSUB 4000
100 GOTO 40

1000 LET a$=INKEY$
1010 IF a$="" THEN RETURN
1020 IF a$="Z" THEN GOTO 2000
1030 LET k=k+1
1040 IF k>8 THEN LET k=1
1050 RETURN

2000 LET xL=x*8+4
2010 LET yL=175-y*8-4
2015 GOSUB 2500
2050 PLOT xL,yL
2060 LET dx=0
2070 IF v(k)=1 THEN LET dx=255-xL
2080 IF v(k)=-1 THEN LET dx=-xL
2090 LET dy=0
2100 IF w(k)=1 THEN LET dy=-yL
2120 IF w(k)=-1 THEN LET dy=175-yL
2130 IF v(k)*w(k)=0 THEN GOTO 2200
2140 IF ABS dx<ABS dy THEN
    LET dy=ABS dx*SGN dy: GOTO 2200
2150 LET dx=ABS dy*SGN dx
2200 DRAW dx,dy
2210 PLOT xL,yL
2215 BEEP ,01,10
2220 DRAW OVER 1;dx,dy

2230 IF h=0 THEN RETURN
2235 LET mx=b*8+4: LET my=175-a*8-4
2240 FOR i=1 TO RND*5+20
2250 PLOT mx,my
2260 LET dx=10-RND*20
2270 IF mx+dx>255 OR mx+dx<0 THEN
    GOTO 2330

```

```

2280 LET dy=10-RND*20
2290 IF my+dy>175 OR my+dy<0 THEN
  GOTO 2330
2300 PLOT mx,my
2310 DRAW dx,dy
2320 BEEP .01,-10
2330 NEXT i
2340 RETURN
2500 LET h=0
2510 LET dy=e-y
2520 LET dx=b-x
2530 IF w(k)*dx<0*v(k)*dy THEN RETURN
2540 IF ABS v(k)*SGN dx<0*v(k)
  OR ABS w(k)*SGN dy<0*w(k) THEN RETURN
2550 LET h=1
2560 RETURN

```

```

3000 IF nb1=0 THEN PRINT AT y,x;" "
3010 LET x=x+v(k)
3020 LET y=y+w(k)
3030 IF x<0 THEN LET x=31
3040 IF x>31 THEN LET x=0
3050 IF y<0 THEN LET y=21
3060 IF y>21 THEN LET y=0
3065 IF ATTR (y,x)<>15 THEN BEEP .1,-10:
  LET nb1=1: GOTO 1030
3070 PRINT AT y,x;m$(k)
3080 LET nb1=0
3090 RETURN

4000 IF RND>1.05-df/20 THEN LET z=z+1
4010 IF z>8 THEN LET z=1
4020 IF nb2=0 THEN PRINT AT a,b;" "
4030 LET a=a+v(z)
4040 LET b=b+w(z)
4050 IF b<0 THEN LET b=31

4060 IF b>31 THEN LET b=0
4070 IF a<0 THEN LET a=21
4080 IF a>21 THEN LET a=0
4085 IF ATTR (a,b)<>15 THEN BEEP .1,-10:
  LET nb2=1: LET z=z+1: GOTO 4010
4090 PRINT AT a,b;w$(r+1)
4100 LET r=NOT r
4110 LET nb2=0
4120 RETURN

```

```

7000 PAPER 1
7010 BORDER 5
7020 INK 7
7030 CLS
7040 PRINT AT 2,5;"L a s e r A t t a c k"
7050 PRINT AT 5,0;"You are in control
  of an"
7060 PRINT "advanced laser attack ship in"
7070 PRINT "pursuit of an enemy craft"
7080 PRINT // "Shoot it down before your
  time"
7090 PRINT TAB 5;"is up !!!!"
7100 INPUT "Select the difficulty level
  - 1 (easy) to 10 (difficult) ?";df
7110 IF df<1 OR df>10 THEN GOTO 7100
7120 CLS
7130 RETURN

7500 LET t=(PEEK 23672+256*PEEK 23673+
  65536*PEEK 23674)/5
7510 PRINT AT 0,0; INT t/10;" "
7520 RETURN

7900 IF RND>.1+df/50 THEN RETURN
7910 PRINT AT RND*21,RND*31; INK 6;"x"
7920 RETURN

8000 DIM w(8): DIM v(8)
8010 DIM s$(8,8): DIM r$(8,8)
8020 DATA 0,1,1,1,1,0,1,-1,0,-1,-1,-1,
  -1,0,-1,1
8030 FOR i=1 TO 8
8040 READ w(i),v(i)
8050 NEXT i

8060 LET k=1
8070 LET x=20
8080 LET y=10
8090 LET v=v(k)
8100 LET w=w(k)
8110 LET s$(1)="00011000"
8120 LET s$(2)="00111100"
8130 LET s$(3)="01111110"
8140 LET s$(4)="11111111"
8150 LET s$(5)="00111100"
8160 LET s$(6)="00111100"

```

```

8170 LET s$(7)="00111100"
8180 LET s$(8)="00111100"
8190 LET r$(1)="11111000"
8200 LET r$(2)="11110000"
8210 LET r$(3)="11111000"
8220 LET r$(4)="11111100"
8225 LET r$(5)="10111110"
8230 LET r$(6)="00011111"
8240 LET r$(7)="00001110"
8250 LET r$(8)="00000100"
8260 FOR i=1 TO 8
8270 LET a=0: LET b=0: LET c=0
8275 LET d=0: LET e=0: LET f=0
8280 FOR j=1 TO 8
8290 LET a=a*2+VAL (s$(i,j))
8300 LET b=b*2+VAL (s$(9-j,i))
8310 LET c=c*2+VAL (s$(j,i))
8320 LET d=d*2+VAL (r$(i,j))
8330 LET e=e*2+VAL (r$(9-j,i))
8340 LET f=f*2+VAL (r$(9-i,9-j))
8345 GOSUB 7900
8380 NEXT j
8390 POKE USR "a"+i-1,a
8400 POKE USR "b"+8-i,a
8410 POKE USR "c"+i-1,b
8420 POKE USR "d"+i-1,c
8430 POKE USR "e"+i-1,d
8440 POKE USR "f"+8-i,d
8450 POKE USR "g"+i-1,e
8460 POKE USR "h"+i-1,f
8490 NEXT i

8500 LET m$="c|h|b|f|d|e|a|g"
8510 LET a=10
8520 LET b=10
8530 LET z=INT (RND*8)+1

8600 POKE USR "i"+0,BIN 00100000
8610 POKE USR "i"+1,BIN 01000010
8620 POKE USR "i"+2,BIN 00100101
8630 POKE USR "i"+3,BIN 00011000
8640 POKE USR "i"+4,BIN 00011000
8650 POKE USR "i"+5,BIN 10100100
8660 POKE USR "i"+6,BIN 01000010
8670 POKE USR "i"+7,BIN 00000100
8680 POKE USR "j"+0,BIN 00001110

```

```

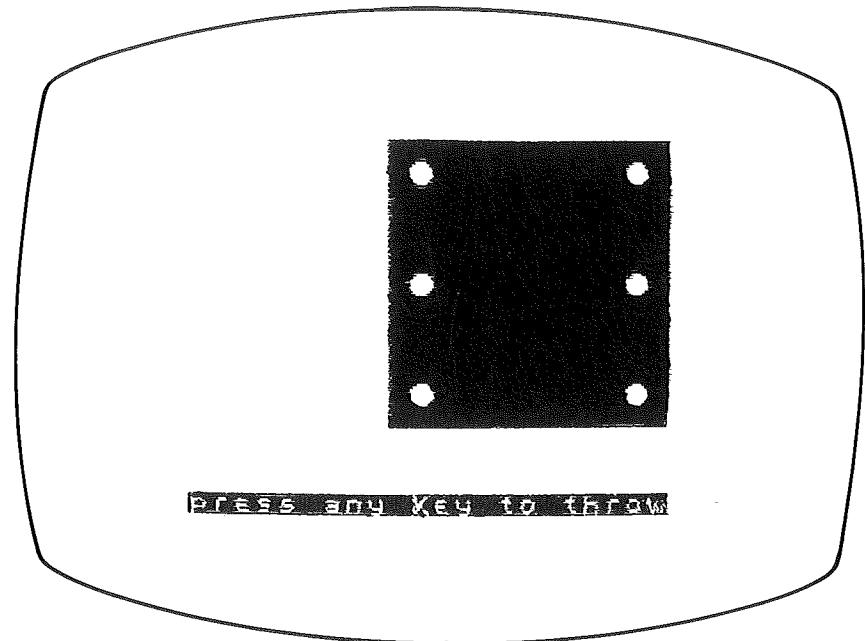
8690 POKE USR "j"+1,BIN 10001000
8700 POKE USR "j"+2,BIN 10001000
8710 POKE USR "j"+3,BIN 11111000
8720 POKE USR "j"+4,BIN 00011111
8730 POKE USR "j"+5,BIN 00001001
8740 POKE USR "j"+6,BIN 00001001
8750 POKE USR "j"+7,BIN 00111000
8760 LET w$="c|h|b|f|d|e|a|g"
8770 LET r=0
8780 INK 7
8790 LET nb1=0
8800 LET nb2=0

8810 POKE 23674,0
8820 POKE 23673,0
8830 POKE 23672,0
8840 LET h=0
8900 RETURN

9000 IF h<>1 THEN GOTO 9500
9010 PRINT AT 21,0;"You did it !!!!"
9020 GOTO 9600
9500 PRINT AT 21,0;"Your time is up"
9600 INPUT "Another game y/n ?";a$
9610 IF a$(1)="y" THEN RUN
9620 PAPER 7
9630 INK 0
9640 BORDER 7
9650 CLS

```

3 **Spectrum Dice**



Before the days of the micro, family games usually meant one of two things – card games or board games that involved dice. In our family we often could not find the dice and we spent ages hunting through drawers and cupboards before we could start our game. Equally often, in the excitement of the game, the dice would end up rolling over the floor and our game would be interrupted as we retrieved it from dark corners.

You may think there's no place for your old games of Ludo and Monopoly now you have a Spectrum to absorb you, but think again. They are actually very enjoyable games for lots of players especially if you don't have to spend so much time hunting for the dice, or worse still, arguing about which way up it actually fell! Such problems can be solved if you let your Spectrum join in the game and take over from the dice.

Of course, your Spectrum Dice can become the centre of a game. You can devise gambling games to play against the computer or against other people. After all, dice have been around for thousands of years so there must be plenty of ideas about how to use them.

However you choose to use the program, it will give you a large clear display on the TV screen – colourful too if you run it on a colour set – and you'll be impressed

by the way it even sounds like a dice rolling over a wooden surface and actually slows down before it comes to a final halt.

How to use the program

Using this program is simplicity itself. Type RUN and, when your Spectrum prompts, just press any key in order to start the dice rolling and then it carries on rolling for a random number of turns and slows down and stops when it is ready. When you've finished with the program press the BREAK key to stop it running.

Typing tips

In the subroutine that starts at line 1000, you'll notice that "[a]" appears a number of times. The square brackets around the letter 'a' indicate that it is a graphics character. Do not type the brackets, instead get into graphics mode, so that you see the G cursor, (you do this by pressing the 9 key with the CAPS SHIFT held down) and then type an 'a'. Remember to type the double quotation marks around it.

In line 2010, notice that you have to type 13 spaces between the quotation marks. Count these carefully as your dice display will not work with the wrong number.

Subroutine structure

| | |
|------|--|
| 10 | Main play loop |
| 1000 | Prints and unprints dots |
| 2000 | Draws yellow square for dice |
| 5000 | Defines dot character [a] and sets colours |
| 6000 | Emits click sound |

Programming details

The essence of a dice program is in generating random numbers. In fact, this program uses random numbers in two ways. Firstly, randomness is used in the conventional way, to determine which face of the dice will show at the next turn – this is done in line 110, which uses the RND function to select 'r', a number between one and six. This information is then used in the printing subroutine (starting at line 1000). The program goes to one of the six line numbers 1100, 1200, 1300, 1400, 1500, 1600, according to the value of 'r'. At 1100 one dot is printed, at 1200 two dots are printed and so on.

The other use of the random number generator is to give the Spectrum Dice realistic suspense. When a human throws a dice it will turn just a few times or quite a number of times and before it actually stops it will slow down. Your Spectrum Dice copies both the features by incorporating lines 60 to 80. Another random number, 't', with a value between 6 and 16 is selected. This governs the number of

turns the dice makes and each time it rolls over the pause before the dots reappear lengthens.

This program makes use of both colour and sound. Colour is largely looked after in lines 5080 to 5120. However, the colour that the dots will be printed in, red, is set in line 90. The dots are actually made to disappear by overprinting them in the dice's background colour (yellow). This colour is determined in line 2010. You may have to listen fairly carefully to detect the clicking sound the dice makes as it rolls over. This is because the noise comes from the speaker on your Spectrum which is hidden on its underside. Lines 6000 to 6020 are responsible for this very dice-like sound.

Scope for improvement

This program could be incorporated into many self-contained games. You could devise a gambling game where bets were placed on which face of the dice would show.

Program

```

5 REM Spectrum dice

10 GOSUB 5000
20 GOSUB 2000
30 INPUT " "
40 PRINT AT 20,0; PAPER 0; INK 6;
"press any key to throw"
50 PAUSE 0
60 LET t=RND*10+5
70 FOR i=1 TO t
80 PAUSE 1+i
90 INK 6
100 GOSUB 1000
110 LET r=INT (RND*6)+1
120 INK 2
130 GOSUB 1000
140 NEXT i
150 GOTO 50

1000 GOSUB 6000
1010 GOTO 1000+r*100
1100 PRINT AT 10,15;"[a]""
1110 RETURN
1200 PRINT AT 5,10;"[a]""
1210 PRINT AT 15,20;"[a]""
1220 RETURN

```

```

1300 GOSUB 1100
1310 GOTO 1200
1400 PRINT AT 5,20;"CaJ"
1410 PRINT AT 15,10;"CaJ"
1420 GOTO 1200
1500 GOSUB 1400
1510 GOTO 1100
1600 PRINT AT 10,10;"CaJ"
1610 PRINT AT 10,20;"CaJ"
1620 GOTO 1400

2000 FOR i=1 TO 13
2010 PRINT AT 3+i,9; PAPER 6;""
    " REM 13 spaces
2020 NEXT i
2030 LET r=1
2040 GOTO 1000

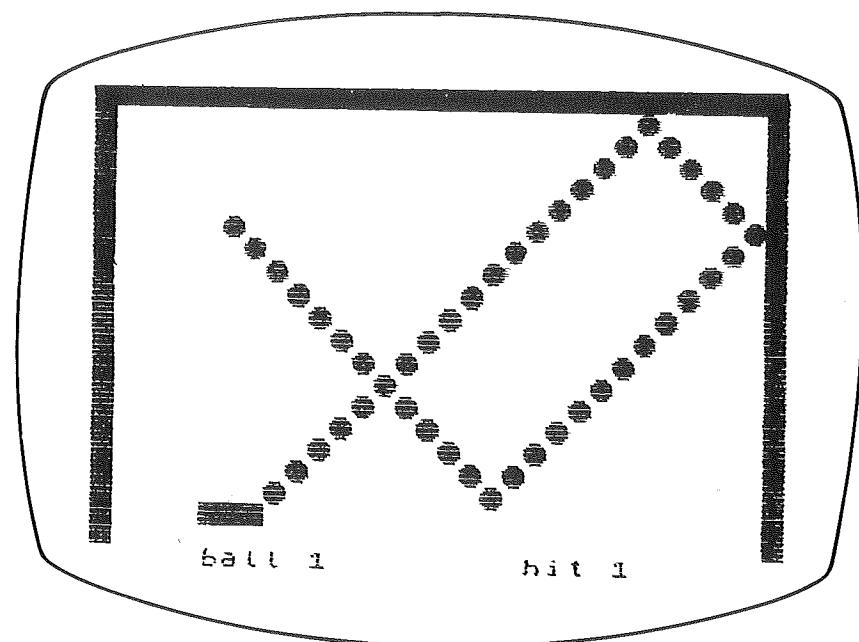
5000 POKE USR "a"+0,BIN 00111100
5010 POKE USR "a"+1,BIN 01111110
5020 POKE USR "a"+2,BIN 11111111
5030 POKE USR "a"+3,BIN 11111111
5040 POKE USR "a"+4,BIN 11111111
5050 POKE USR "a"+5,BIN 11111111
5060 POKE USR "a"+6,BIN 01111110
5070 POKE USR "a"+7,BIN 00111100
5080 INK 2
5090 PAPER 0
5100 CLS
5110 PAPER 6
5120 BORDER 0
5130 RETURN

6000 OUT 254,16
6010 OUT 254,0
6020 RETURN

```

4

Rainbow Squash



This is a colourful version of the popular computer squash game – on a colour TV you'll find yourself playing on a white, red and yellow court with a blue border. It is also enhanced by the addition of sound – a cheery beep every time the ball bounces either on the sides of the court or against the bat and a dismal tone every time a ball goes out of play. Its other feature is that as you improve in skill the game gets more difficult and if you then start to get worse it gets easier. This means that your Spectrum will always give you a challenge that is suited to your ability – which makes it the perfect partner.

How to play

At the start of the game the bat is at the bottom of the screen and in the centre. You control the left and right movement of the bat by pressing the appropriate arrow keys (5 and 8). Every time you make two hits in succession the position of the bat changes – it moves nearer to the top of the screen – which makes returning the ball more difficult. If you then miss a shot the bat will move back one position, making it easier. You score a point for every hit and you will be served a total of 10

balls. Information about the number of balls played and hits scored is displayed on the screen continuously.

Typing tips

The graphics characters used for the bat are solid blocks, produced by pressing CAPS SHIFT and the 8 key while in graphics mode. Notice that in line 110 you type double quotes, followed by one space, followed by three of these solid blocks, followed by another space and finally double quotes. In lines 170 and 470 five blank spaces, between double quotes, are required to blank out the bat.

Subroutine structure

```

5 Sets colours and initialises variables
100 Main loop - prints bat and score line
200 Gets movements of bat
300 Moves balls and controls bouncing off walls
400 Controls bounce against bat and movement of bat up screen
500 Prints walls of court
800 Sets colours of court
930 Defines graphics character for ball
2000 End of game - prints final score, offers another game and re-runs or
restores screen display

```

Programming details

There are some interesting uses of colour commands in this program. In lines 110 and 360 you'll find "PAPER 8". Colour 8 is *transparent* in the sense that it allows whatever colour is already present on the screen to show through. So by using "PAPER 8" in both the bat and ball printing routines, the background colour is unaltered. The ball graphic defined at subroutine 930 is one that you may find useful in your own programs.

Program

```

1 REM Rainbow squash

5 BORDER 1; PAPER 7; INK 0
10 LET h=0; LET ht=0; LET d=19;
LET ball1=0
15 GOSUB 800; GOSUB 500
20 LET ball1=ball1+1
30 IF ball1>10 THEN GOTO 2000
40 LET a=5
50 LET b=5

```

```

60 LET v=1
70 LET w=1
80 LET x=10; LET y=d
90 INPUT " "
100 PRINT AT 21,5; "ball "; ball;
110 PRINT AT y,x; PAPER 8; INK 0;
" [^8][^8][^8] "
120 PRINT AT 21,20; "hit "; ht
130 GOSUB 300
140 GOSUB 200
150 IF b+w>y THEN LET y=d; GOTO 110
160 BEEP 1,0
170 PRINT AT y,x; PAPER 8; "      ";
REM 5 spaces
180 PRINT AT b,a; PAPER 8; " "
190 IF d<19 THEN LET d=d+1
194 LET h=0
195 GOTO 20
200 LET a$=INKEY$
210 IF a$="5" AND x>0 THEN LET x=x-1
220 IF a$="8" AND x<27 THEN LET x=x+1
230 RETURN
300 PRINT AT b,a; PAPER 8; INK 8; " "
310 LET a=a+v
320 LET b=b+w
330 IF a=30 OR a=1 THEN LET v=-v;
BEEP .1,.15
340 IF b=1 THEN LET w=w; BEEP .1,.15
350 IF b+w=y THEN GOTO 400
360 PRINT AT b,a; PAPER 8; "[a]"
370 RETURN
400 LET r=a-x
410 IF r<1 OR r>3 THEN GOTO 360
420 LET w=w
430 BEEP .1,.15
440 LET h=h+1; LET ht=ht+1
450 IF h>2 THEN GOTO 360
460 LET h=0; LET d=d-1
470 PRINT AT y,x; PAPER 8; "      ";
REM 5 spaces
490 GOTO 300

```

```

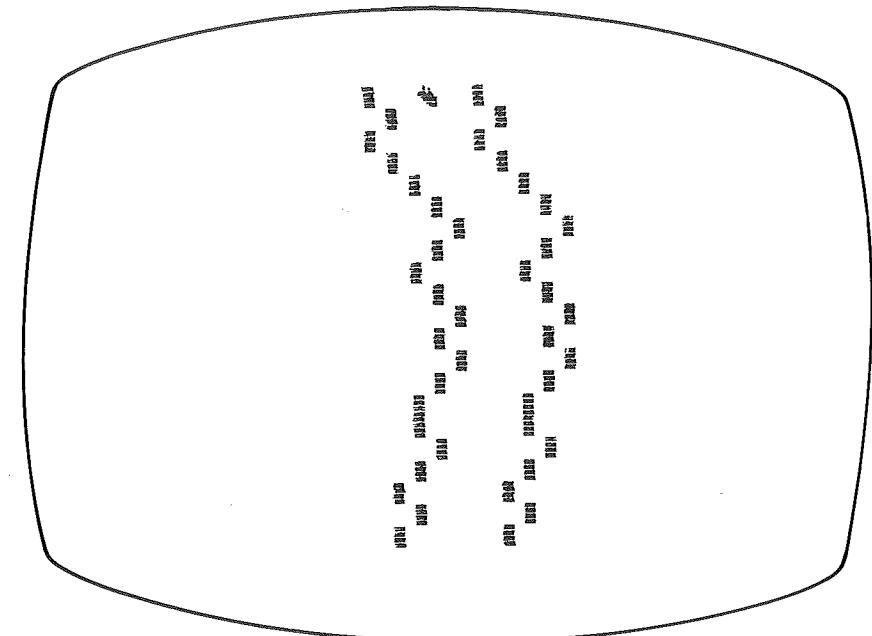
500 FOR i=0 TO 31
510 PRINT AT 0,i; PAPER 0;" "
520 NEXT i
530 FOR i=0 TO 20
540 PRINT AT i,0; PAPER 0;" ");
    AT i,31;" "
550 NEXT i
560 RETURN

800 PRINT AT 7,0;
810 LET c=2: GOSUB 850
820 LET c=6: GOSUB 850
830 GOSUB 930
840 RETURN
850 FOR i=1 TO 7*32
860 PRINT PAPER c;" ")
870 NEXT i
880 RETURN
930 POKE USR "a"+0,BIN 00111100
940 POKE USR "a"+1,BIN 01111110
950 POKE USR "a"+2,BIN 11111111
960 POKE USR "a"+3,BIN 11111111
970 POKE USR "a"+4,BIN 11111111
980 POKE USR "a"+5,BIN 11111111
990 POKE USR "a"+6,BIN 01111110
1000 POKE USR "a"+7,BIN 00111100
1010 PAPER 7
1020 RETURN

2000 INK 0
2010 PAPER 7
2030 CLS
2040 PRINT AT 10,10; "You scored"
2050 PRINT AT 12,14;ht
2060 INPUT "Another game y/n ? ";a$
2070 IF a$(1)="y" THEN RUN
2080 CLS: BORDER 7
2090 PRINT "Bye"

```

Bobsleigh



In this game you can choose to steer your red bobsleigh down a random course that is easy to manoeuvre or one that is difficult (there are actually five levels of difficulty which govern the width of the course). If you crash you'll hear a dismal tone and that round of the game is over. Play this game to see how adept you are at keeping on course.

How to play

The bobsleigh starts off at the top of the course and the course automatically moves past it. You have to steer the bobsleigh using the right and left arrow keys to ensure that you do not crash into the edges of the course. At the beginning of the game you have to select the difficulty level for the game. This governs the width of the course with 1 producing the widest, and therefore easiest, and 5 the narrowest, and most difficult.

Typing tips

This game involves a user-defined graphics character and a supplied graphics

character. Both are listed within square brackets. Do not type in the brackets. Instead type in the appropriate letter, 'b', or number '5' in graphics mode. Notice that you have to type two apostrophes at the ends of lines 1320 and 3100.

Subroutine structure

```

500 Define bobsleigh graphics character
1000 Print track and bobsleigh
2400 Scroll screen
5000 Move bobsleigh
7000 Title frame
8000 Win/lose messages
9000 End of game

```

Programming details

The impression of the bobsleigh moving down the course is actually achieved by the course *scrolling* up the screen past the bobsleigh which only moves to left and right and is at a fixed vertical position.

In line 5320 ATTRibute is used to test whether or not the bobsleigh has hit the side wall. This is done simply by testing what colour is present at the next position that the bobsleigh will be printed at. If the colour is not white then you've crashed into the wall.

Scope for improvement

If you find this game too fast-moving you can slow it down by introducing a PAUSE into subroutine 2400.

Program

```

5 REM Bobsleigh

20 GOSUB 7000
500 POKE USR "b"+0,BIN 00100000
510 POKE USR "b"+1,BIN 00101000
520 POKE USR "b"+2,BIN 11101000
530 POKE USR "b"+3,BIN 11111100
540 POKE USR "b"+4,BIN 01111110
550 POKE USR "b"+5,BIN 00111110
560 POKE USR "b"+6,BIN 00011110
570 POKE USR "b"+7,BIN 00001110
800 LET yb=1

1000 REM print track

```

```

1010 PAPER 7: INK 7
1020 CLS
1050 LET x=(RND*10)+5
1060 LET xb=x+2
1100 FOR y=1 TO 20
1140 PRINT AT y,x; INK 0; "E5J";
      AT y,x+d; "E5J"
1150 LET x=x+(SGN (RND-.5))
1160 IF x>31 THEN LET x=31
1170 IF x<0 THEN LET x=0
1200 NEXT y
1220 PRINT AT yb,xb; INK 2; "EbJ"
1230 PAUSE 50
1250 FOR z=1 TO 20
1260 LET x=x+(SGN (RND-.5))
1270 IF x>31 THEN LET x=31
1280 IF x<0 THEN LET x=0
1310 POKE 23692,41
1320 PRINT AT 21,x; INK 0; "E5J";
      AT 21,x+d; "E5J"; //
2000 GOSUB 5000

2400 NEXT z
3000 FOR z=1 TO 20
3100 PRINT AT 21,1 //
3150 GOSUB 5000
3200 NEXT z
3500 GOSUB 8000

5000 REM move bobsleigh
5100 LET a$=INKEY$
5150 PRINT AT yb-1,xb;" "
5200 IF a$="5" THEN LET xb=xb-1
5210 IF a$="8" THEN LET xb=xb+1
5320 IF ATTR (yb,xb)=56 THEN GOTO 8500
5330 PRINT AT yb,xb; INK 2; "EbJ"
5500 RETURN

7000 INK 0
7010 PAPER 7
7020 PRINT AT 1,5;"B o b s l e i g h"
7030 PRINT AT 5,0;"You must steer your
      bobsleigh"
7040 PRINT "down a dangerous course"
7050 INPUT "Select the difficulty level -"
      // "from 1 (easy) to 5 (difficult) ?";d

```

```

7060 IF d<1 OR d>5 THEN GOTO 7050
7070 LET d=9-d
7080 RETURN

8000 PRINT AT 20,1; INK 0;
  "Congratulations, you made it"
8100 GOTO 9000
8500 PRINT AT 20,1; INK 0; "You crashed"
8510 BEEP 1,-10

9000 INPUT "another game (y/n)";b$
9020 IF b$(1)="y" THEN RUN
9500 INK 0
9510 PAPER 7
9520 CLS

```

Spectrum Smalltalk

Tell me about your problems

I have an awkward computer

Do computers worry you?

Not always

Give me a particular example

It crashed my program

YOUR PROGRAM?

Do you ever find yourself *talking* to your Spectrum? Well, if you do you may be disappointed that it never answers back. This program, however, changes all that and gives your Spectrum the chance to start a conversation with you. Although it may not be able to rival the agony aunts of the glossy magazines, your Spectrum is anxious to hear about your problems – and has some comments to offer.

Coping with the syntax of the English language is a very complicated problem which this program has to contend with. Programs like this one have been developed in order to extend our knowledge of how language works and how humans identify the key components of conversations. While these serious purposes are usually the province of *artificial intelligence* it is possible to have a good deal of fun trying to conduct a dialogue with your Spectrum.

How to use this program

The computer opens each conversation in the same way – by inviting you to tell it your problems. You can give any reply that you wish to and after a few moments' delay your Spectrum will respond. Try to say more than just 'Yes' or 'No' when you make further responses but equally, don't say too much at a time. If you type

about a lineful each time you ought to be able to keep a reasonable *conversation* going.

Typing tips

Both when typing this program in and when using it, do be careful about your spelling. If you type in either the initial program or subsequent responses with misspellings the computer won't recognise your message and you won't receive any sensible answers. The apostrophe is the only punctuation mark that should be used in dialogues with the Spectrum and, while typing in the program, notice its use not only within words but also to throw lines of space on the screen – as in lines 2202, 2220 and 9830. Another point to notice in the listing is the occurrence of '(TO ' and ' TO)' for string slicing.

Subroutine structure

| | |
|------|------------------------------|
| 20 | Main program loop |
| 1000 | Initial message and set up |
| 2000 | Input human's sentence |
| 3000 | Divides sentences into words |
| 3600 | Changes tense/pronouns |
| 3800 | Tense/pronouns data |
| 5000 | Finds keywords in sentence |
| 6000 | Keywords data |
| 7000 | Keyword responses |
| 9800 | Prints computer's response |
| 9900 | Requests sensible input |

Programming details

This program works by taking a sentence and splitting it down into individual words and then responding according to a list of keywords that it searches for in each sentence. So if, for example, your sentence contains the word 'why', the response 'Some questions are difficult to answer' will always be given by the computer. When the computer fails to find a specific reply to a sentence one of a number of responses is selected at random.

Although this technique sounds simple, the actual details of the program are really quite tricky as, amongst other things, the computer has to deal with tense changes and with the syntax of pronouns. It is therefore quite a difficult program to write or to modify extensively. Equally, despite the apparent simplicity of its underlying technique, it succeeds in making plausible responses on a surprising number of occasions.

Scope for improvement

If you wish to add to the list of keywords that the computer recognises, you need to notice how, in subroutine 6000, the keywords are paired with the line number of the subroutine that responds to them. Also it is important to be aware of the priorities assigned to each keyword. If two keywords are present in a sentence then the one first in the list will be acted upon.

Program

```

10 REM Spectrum smalltalk.

20 GOSUB 1000
30 GOSUB 2000
40 GOSUB 3000
50 GOSUB 5000
60 IF num<>0 THEN GOSUB num
70 GOSUB 9800
80 GOTO 30

1000 CLS
1010 PRINT AT 1,2;"Hi! My name is
ZX Spectrum"
1030 PRINT
1040 PRINT "I would like you to talk to me"
1050 PRINT "but I don't have ears so will"
1060 PRINT "you type sentences on me"
1070 PRINT "keyboard"
1074 PRINT
1075 PRINT "Don't use any punctuation apart"
1076 PRINT "from apostrophes which are"
1078 PRINT "important"
1080 PRINT
1110 PRINT "When you have finished typing"
1115 PRINT "press ENTER"
1120 PRINT AT 18,0;BRIGHT 1;
      "Tell me about your problems"
1130 LET r$=""
1140 LET m$=""
1150 LET d$=""
1160 DIM r$(3,32)
1170 LET r$(1)="Please go on"
1180 LET r$(2)="I'm not sure I
understand you"
1190 LET r$(3)="Tell me more"
1200 DIM i$(3,40)

```

```

1210 LET i$(1)="Let's talk some more
      about your"
1220 LET i$(2)="Earlier you spoke of your"
1230 LET i$(3)="Does that have anything
      to do with your"
1235 DIM j$(2,32)
1240 LET j$(1)="Are you just being negative"
1250 LET j$(2)="I see"
1990 RETURN

2000 LET a$=""
2010 LET b$=INKEY$
2015 POKE 23692,255
2020 IF b$="" THEN GOTO 2010
2025 IF INKEY$<>"" THEN GOTO 2025
2030 IF CODE(b$)=13 THEN GOTO 2200
2040 IF CODE(b$)=12 AND a$<>"" THEN LET
      a$=a$(1 TO LEN a$-1): GOTO 2090
2050 IF CODE(b$)<32 OR CODE(b$)>126 THEN
      GOTO 2000
2060 LET a$=a$+b$
2090 PRINT AT 20,0;a$;" "
2100 GOTO 2010
2200 IF a$(LEN a$)="" THEN
      LET a$=a$(1 TO LEN a$-1): GOTO 2200
2202 IF a$=r$ THEN PRINT AT 21,0; "You're
      repeating yourself!"'; GOTO 2000
2203 LET r$=a$
2204 IF a$="" THEN GOTO 2000
2205 LET a$=" "+a$
2215 IF CODE(a$(2))<97 THEN
      LET a$(2)=CHR$(CODE(a$(2))+32)
2220 PRINT AT 21,0; ""
2230 RETURN

3000 DIM w(20,2)
3005 LET n=1
3010 LET b=0
3020 FOR i=1 TO LEN a$
3040 IF (a$(i)="" OR a$(i)=",") AND b=0
      THEN LET b=1
3050 IF (a$(i)<>"" OR a$(i)<>",") AND b<=1
      THEN LET w(n,1)=i: LET b=2
3060 IF (a$(i)="" OR a$(i)=",") AND b=2
      THEN LET w(n,2)=i-1: LET n=n+1: LET b=0
3070 NEXT i
3080 LET w(n,2)=LEN a$

```

```

3600 FOR i=1 TO n
3605 RESTORE 3800
3610 READ b$
3620 IF b$="s" THEN GOTO 3690
3630 IF b$<>a$(w(i,1) TO w(i,2)) THEN
      GOTO 3680
3640 READ c$
3650 LET a$=a$(1 TO w(i,1)-1)+c$+
      a$(w(i,2)+1 TO )
3654 LET w(i,2)=w(i,2)+LEN c$-LEN b$
3655 FOR j=i+1 TO n
3660 LET w(j,2)=w(j,2)+LEN c$-LEN b$
3664 LET w(j,1)=w(j,1)+LEN c$-LEN b$
3665 NEXT j
3670 GOTO 3690
3680 READ b$
3685 GOTO 3610
3690 NEXT i
3700 RETURN

3800 DATA "m$","your*","I","you*","i","you"
3810 DATA "mum","Mother","dad","Father"
3820 DATA "dreams","dream","you","Ix",
      "me","you"
3830 DATA "your","m*","myself","yourself*",
      "I'm","you're"
3840 DATA "yourself","myself*",
      "I'm","you're"
3850 DATA "you're","I'm*","am","are"
3870 DATA "i'm","you're"
3880 DATA "were",was"
3885 DATA "are",am"
3890 DATA "s","s"

5000 RESTORE 6000
5010 READ b$,num
5020 IF b$="s" THEN GOTO 5710
5025 FOR i=1 TO n
5030 IF a$(w(i,1) TO w(i,2))<>b$ THEN
      GOTO 5700
5040 LET t$=a$(w(i,2)+1 TO )
5050 RETURN
5700 NEXT i
5705 GOTO 5010
5710 LET num=0
5720 IF num<>0 THEN GOTO 5800

```

```

5730 LET p$=r$(INT (RND*3)+1)
5740 RETURN
5800 LET p$=i$(INT (RND*3)+1)+m$
5900 RETURN

6000 DATA "computer",7000,"machine",7000
6010 DATA "like",7100,"same",7100,
      "alike",7100
6020 DATA "if",7200,"everybody",7300
6024 DATA "can",8200,"certainly",8250
6025 DATA "how",8100,"because",8150
6026 DATA "always",7800
6030 DATA "everyone",7300,"nobody",7300
6034 DATA "was",7500
6035 DATA "I*",8300
6040 DATA "no",7400
6060 DATA "your*",7600
6070 DATA "you're*",8500,"you*",8650
6110 DATA "hello",8300,"maybe",8350
6120 DATA "my*",8370,"no",8420
6130 DATA "yes",8250,"why",8450
6140 DATA "perhaps",8350,"sorry",8400
6160 DATA "what",8450
6900 DATA "s",0

7000 LET p$="Do computers worry you?"
7010 RETURN
7100 LET p$="In what way ?"
7110 RETURN
7200 LET p$="Why talk of possibilities"
7210 RETURN
7300 LET p$="Really "+b$+" ?"
7310 RETURN
7400 IF i>n THEN GOTO 7450
7410 LET i=i+1
7420 IF a$(w(i,1) TO w(i,2))="one" THEN
      LET b$=b$+" one":GOTO 7300
7450 LET p$=j$(INT (RND*2)+1)
7460 RETURN
7500 IF i>n THEN GOTO 9900
7510 LET i=i+1
7515 IF i>n THEN GOTO 5720
7520 IF a$(w(i,1) TO w(i,2))<>"your" THEN GOTO 7550
7530 LET p$="What if you were "+
      a$(w(i,2)+1 TO )+" ?"
7540 RETURN

```

```

7550 IF a$(w(i,1) TO w(i,2))<>"I*" THEN
      GOTO 5720
7560 LET p$="Would you like to believe I
      was "+a$(w(i,2)+1 TO )
7570 RETURN

7600 LET i=i+1
7605 IF i>n THEN GOTO 7450
7610 IF a$(w(i,1) TO w(i,2))=="Mother" THEN
      GOTO 7700
7620 IF a$(w(i,1) TO w(i,2))=="Father" THEN
      GOTO 7700
7630 IF a$(w(i,1) TO w(i,2))=="sister" THEN
      GOTO 7700
7640 IF a$(w(i,1) TO w(i,2))=="brother" THEN
      GOTO 7700
7650 IF a$(w(i,1) TO w(i,2))=="wife" THEN
      GOTO 7700
7660 IF a$(w(i,1) TO w(i,2))=="husband" THEN
      GOTO 7700
7670 IF a$(w(i,1) TO w(i,2))=="children" THEN
      GOTO 7700
7680 IF LEN t$>10 THEN LET m$=t$
7690 LET p$="your "+t$+" ?"
7695 RETURN
7700 LET p$="Tell me more about your family"
7710 RETURN
7800 LET p$="Give me a particular example"
7810 RETURN
7900 LET i=i+1
7905 IF i>n THEN LET p$="Am I what ?":RETURN
7920 LET p$="Why are you interested in
      whether I am "+a$(w(i,1) TO )+" or not?"
7930 RETURN
8000 LET p$="Do you think you are "+
      a$(w(i,2) TO )
8030 RETURN
8100 LET p$="Why do you ask ?"
8110 RETURN
8150 LET p$="Tell me about any other reasons"
8160 RETURN
8200 LET i=i+1
8205 IF i>n THEN LET p$="What ?":RETURN
8210 IF a$(w(i,1) TO w(i,2))=="I*" THEN
      LET p$="Do you believe I can"+
      a$(w(i,2)+1 TO )+" ?":RETURN

```

```

8220 IF a$(w(i,1) TO w(i,2))="you*" THEN
  LET p$="Do you believe you can "+  

  a$(w(i,2)+1 TO )+" ?":RETURN
8230 GOTO 5720
8250 LET p$="You seem very positive"
8260 RETURN
8300 LET p$="Pleased to meet you - let's
  talk about your problems"
8310 RETURN
8350 LET p$="Could you try to be more
  positive"
8360 RETURN
8370 LET p$="Why are you concerned about
  my "+t$  

8380 RETURN
8400 LET p$="You don't have to apologise
  to me"
8410 RETURN
8450 LET p$="Some questions are difficult
  to answer... "
8460 RETURN
8500 LET i=i+1
8505 IF i>n THEN GOTO 5720
8510 LET p$="I am sorry to hear that you are"
  +a$(w(i,1) TO w(i,2))
8520 IF a$(w(i,1) TO w(i,2))="sad" THEN
  RETURN
8530 IF a$(w(i,1) TO w(i,2))="unhappy" THEN
  RETURN
8540 IF a$(w(i,1) TO w(i,2))="depressed" THEN
  RETURN
8550 IF a$(w(i,1) TO w(i,2))="sick" THEN
  RETURN
8560 LET p$="How have I helped you to be "+  

  a$(w(i,1) TO w(i,2))
8570 IF a$(w(i,1) TO w(i,2))="happy" THEN
  RETURN
8580 IF a$(w(i,1) TO w(i,2))="elated" THEN
  RETURN
8590 IF a$(w(i,1) TO w(i,2))="glad" THEN
  RETURN
8600 IF a$(w(i,1) TO w(i,2))="better" THEN
  RETURN
8610 LET p$="Is it because you are "+  

  a$(w(i,1) TO )+" you would like
  to talk to me ?"

```

```

8620 RETURN
8650 IF i=1 THEN GOTO 8660
8654 IF a$(w(i-1,1) TO w(i-1,2))="are*" THEN
  GOTO 8000
8655 LET i=i+1
8656 IF i>n THEN GOTO 9900
8660 IF a$(w(i,1) TO w(i,2))="are*" THEN
  GOTO 8500
8670 IF a$(w(i,1) TO w(i,2))="want" OR
  a$(w(i,1) TO w(i,2))="need" THEN
  LET p$="what would it mean if you got "
  +a$(w(i,2)+1 TO ): RETURN
8675 IF a$(w(i,1) TO w(i,2))="think" THEN
  LET p$="do you really think so": RETURN
8680 IF a$(w(i,1) TO w(i,2))="can't" OR
  a$(w(i,1) TO w(i,2))="cannot" THEN
  LET p$="How do you know you can't "
  +a$(w(i,2)+1 TO ): RETURN
8690 IF a$(w(i,1) TO w(i,2))="feel" THEN LET
  p$="tell me more about how you feel":  

  RETURN
8700 GOTO 5720
8800 IF i-1<1 THEN GOTO 8805
8804 IF a$(w(i,1) TO w(i,2))="am" THEN
  GOTO 7900
8805 LET i=i+1
8810 IF i>n THEN LET p$="What am I ?":  

  RETURN
8820 IF a$(w(i,1) TO w(i,2))="am" THEN
  LET p$="Why do you think so ?": RETURN
8830 LET p$="Is that what you think of me !"
8840 RETURN
9800 FOR j=1 TO LEN p$  

9810 IF p$(j)<>"*" THEN PRINT BRIGHT 1;p$(j):  

9820 NEXT j
9830 PRINT AT 21,0 ////
9840 RETURN
9900 LET p$="Please talk sensibly !"
9910 RETURN

```

Part 4

Adventures with the Spectrum

by Allan Scott

Chapter Fifteen

A Sense of Adventure

There is another type of game, not considered in the previous section, that I believe deserves a section to itself. This kind of game often lacks the visual seduction of the fast graphics-based arcade game, and usually involves a series of simple printouts on screen, yet it can be more exciting, more involving, and more demanding than even the best arcade game because it demands fast thinking rather than fast reflexes.

Adventure games began their computerised lives on massive disk-based systems, but ironically enough, it's now possible to buy many of these golden oldies in versions for 48K computers! That is partly a measure of what has happened to computers in the intervening years, and partly a measure of the increasing ingenuity of programmers. Either way, the news is good for adventure fans.

For those of you who haven't played an adventure game, and who may be wondering what all this is about, a short history lesson may be in order.

War games

The adventure business grew, originally, out of wargaming. Wargaming involves the use of models, and model landscapes, to represent armies and battlefields. Generally real battle situations are used, and the players refight them using their own military skill and imagination. Each figure on the board represents an agreed number of men, and when two groups of figures meet the combat is fought out with dice (for the chance element) and a strict set of rules. Three hundred barbarian tribesmen stand little chance against a disciplined force of one hundred Roman legionaries, for instance.

Wargaming is educational and entertaining, but it can be a bit slow – in fact it almost cries out for computerisation. Strangely enough, though, computer adventure games owe far more to a direct descendant of 'respectable' wargaming called *Dungeons and Dragons*, which takes wargaming into the realms of fantasy and magic. D & D, as addicts call it, can involve as many as thirty players in strange and mystical adventures, with a cast list including anything from bronze dragons to malicious goblins. The object of the exercise is to win treasure, magical weapons and objects, and undying glory. D & D isn't really a board game. The only man with a complete map is the dungeon master, who acts as referee, tells the players the results of their actions ('I look round the corner' – 'Your head is bitten off by a dire wolf') and keeps track of such delightful game residents as random

monsters. If you've never played D & D it's hard to imagine the fascination of all this, but even seasoned players can become completely engrossed in the game. For a few hours, they are transported to a world of caves and tunnels where magic works, creatures out of myths are real, and almost anything can happen. They even identify with their game characters, talking of their game adventures almost as if they'd really happened. (I speak from personal experience!) This caused such a stir in the Bible belt of the United States that several ministers spoke out against D & D as a form of Satanism. (I suppose they were confused by the magical elements of the storyline!)

In D & D, individual dungeons are created by individual dungeon masters according to a massive book of rules laid out by the game's creators, Gygax and Arneson. Most of the fun of the game depends on the dungeon master – if his imagination or his planning go awry, so does the game. So it's not surprising some of the early computer game designers, notably Scott Adams, saw D & D as a valuable base for a computer game. For the first time, an individual player could pit his wits against a totally impartial dungeon master who would provide all the situations, all the rooms, all the treasures, and all the random monsters, according to a completely fair set of pre-programmed rules.

It works. It works well. And it's now immensely popular.

What's it like?

Most adventure games involve pure text – no noisy flying saucers, no laser sights swinging across the screen, no swooping smart bombs. They are less frenetic, and many allow you to take a considerable time about making decisions. The computer describes a place and/or situation in words. You are invited to enter a command describing what you intend to do about it – GO WEST, or PICK UP SWORD, or KILL RHINOCEROS, for instance. The computer is programmed with a 'vocabulary' of words it can recognise, and will respond to commands it 'understands' by carrying out the action within the game that you asked it to perform. If you try to do something that isn't possible within the terms of the game, a message will appear to tell you so. For instance:

You are chained to a stone wall in a deep, cold dungeon. It is very dark, but a patch of light from above glints in the puddle rapidly forming at the centre of the chamber, and the sound of dripping water tells you that within minutes your cell will be flooded out.

What do you do?

LOOK

There is a door to the west.

You can just see a key glinting in the light from above.

What do you do?

GET KEY

You can't reach the key

INVENTORY

You are carrying nothing at all.

You are wearing a leather jerkin, rough leather trousers, and boots.

EXAMINE FLOOR

There is nothing special to see.

EXAMINE WALL

The wall is made of old, crumbling stone.

You hear a scuttling sound ...

HELP

Sorry, I don't understand that.

The scuttling is getting louder ...

PULL CHAIN

You tear the chain out of old, crumbling stone.

The room is full of rats.

What now?

As you can see, the player here responds to the descriptions given by the computer with two-word commands (or sometimes one-word commands). The computer recognises these (or not, as the case may be – this one wasn't programmed to HELP!) and responds appropriately. In this case the player may have been just a bit too slow – wonder if he'll escape the rats? Some computer games allow much longer commands; in fact you can almost talk to them in English! And as you can see from this example, many adventure games are about solving problems.

Will I like it?

I know that was the title of a book about something quite different, but even *that* could be turned into an adventure game. (You are approaching the bedroom door. What do you do now?) Games like that exist already, and lend some substance to the idea that adventure games are just a form of computerised fantasy that keeps square-eyed addicts awake at night and helps to separate them even further from the real world. This is unfair and far from true. Adventure games are about imagination – the player's imagination as well as the game designer's – and people without imagination won't enjoy them. Not everyone likes this kind of mind-stretching, but before you dismiss adventure gaming altogether, consider some of the possibilities.

With adventure games, there are no limits. They can be set anywhere it's

possible to write about, from the stone age to the far future, from Earth in 1985 to Alpha Centauri in 19850, from caves under the ground to the centre of a white dwarf star. And there's nothing to stop them being set within the framework of an existing story, or poem, or play, or piece of literature. One of the best known examples of this genre is certainly *The Hobbit* from Melbourne House, based on J.R.R. Tolkien's world famous fantasy novel. But an adventure game could be based anywhere, even within a specific historical period, with you, the player, as a specific historical character. With good programming, an adventure game set in the early fifteenth century could teach you more about the Hundred Years War than any number of history textbooks, and entertain you at the same time. It would be almost as good as a time machine! And how about revising your English literature by starting an adventure game in the first act of *Hamlet*? The learning potential isn't confined to the arts. Imagine a game set in a research institute where you can only get from room to room by solving scientific problems. Use any scenario you like, from the masterspy-investigates idea to the if-you-don't-get-to-the-mad-professor's-laboratory-soon-he'll-blow-up-the-world gambit. It's got to be more interesting than your chemistry notes!

One thing that makes games like this simple to prepare is that adventure games don't generally use graphics, at least, not often and not much, though some of the new arcade-type games actually use interactive videodisk players to create continuous animation of everything that happens in the game. This lack of pictures means that the player relies to a great extent on his own imagination and the wording of the game itself, which is not necessarily a bad thing for him and certainly not for the poor old programmer. However, recent trends in both fashion and programming have made pictures more common within adventure games. If this sounds worrying, then you'll be relieved to hear that the same trends have also produced programs that allow you to create pictures quickly and economically! The most notable example of such a program is indisputably *The Dungeon Builder* from Dream Software. We'll be talking more about programming for adventure games later on, but for the moment I want to concentrate on two very important things that must come first – the idea and the storyline.

Chapter Sixteen

Telling the Story

You are lost near high cliffs, frozen and blinded by a violent snowstorm. You stumble, fall, and pass out. When you open your eyes you find yourself under a grey and sunless sky on the brink of an ocean that stretches southwards to the far horizon. To the north a black cliff reaches to touch the clouds, a featureless and unclimbable wall of rock. Slow, oily waves wash bleached white flotsam onto the dry gravel of the beach. Scattered across the beach are white, upstanding sticks that seem to grow unmoved by the wind. To the west, in the lee of the cliff, a dark hall stands. You hear a moaning sound, and something moves in the gravel near your left foot.

Visible exits are east and west.

What do you do?

Well, what would you do? Would you go on and enter your first command? Would you shrug and pick up a book, or turn on the TV?

Adventure games, just like books and TV programmes, are mostly sold on their first few minutes. For instance, if you picked up a story that began with the words 'On Friday the fifth of April, at 10.30 in the evening, David decided to kill Joanna' you might at least be curious. Who is David? Who is Joanna? Why does David want to kill her? And what made him decide to do it at that particular time, on that particular day?

Both that sentence and the paragraph at the beginning of this chapter are examples of a very old literary device that I call the *hook*. Hooks are as old as the ancient Greek poems, the *Iliad* and the *Odyssey*, and I'm pretty sure the first caveman storyteller used them. Milton used one for *Paradise Lost* and you see a very slick modern example of the breed at the front of all those American cops-and-robbers programmes on TV. The job of a hook is to pick up your audience by the scruff of the neck and make them listen to you, because you've aroused their curiosity too much for them to do anything else.

The unanswered question ...

If you look back at that opening paragraph you'll see that it, too, is full of unanswered questions. Where are you? It doesn't appear to be any normal beach! What is the 'flotsam' being washed up by that rather unpleasant sounding sea? What are the 'sticks', and what is in that distant hall? What is the sound – and what is it stirring by your left foot?

The object of that opening sequence is to make the player enter that crucial first

command, and if the pressure can be kept up while two or three more are entered, all the better! By that time your player will be well and truly hooked – interested to see what happens next, interested in solving the problems in his path, interested, above all, in winning.

However, even though the hook is undoubtedly the way to start an adventure, it isn't the place to start writing one. In fact, I'd recommend you leave it until you've finished everything else.

Drawing up an outline

When a professional writer is drawing up a book for publication, he normally starts with an *outline* of what he wants to write about. This doesn't just go for books about computers, it goes for any kind of book, including the novel I used as the source for that first paragraph. There are two reasons for doing it this way. One is to make it clear to any interested buyer what you have in mind. The second, far more important reason is to make it clear to yourself.

The temptation for any would-be writer is always to rush for his pad and his ballpoint (or his typewriter, or his word processor – see Part 5!) and start a frenzied orgy of writing that only stops when he realises that *he* doesn't know what happens next! Most of the best stories in the world have benefited from a good deal of advance planning, even Shakespeare's (after all, he lifted entire stories from other people and transformed them into something better). If you want to write the world's greatest adventure game, *plan* it.

To give you an example, let's follow the development of the game outlined above. In this case, the basic story already exists – it's called *The Ice King*, and it's a tale of supernatural forces from the Viking age, adapted here by kind permission of the author, Michael Scot. Incidentally, if you want to adapt someone else's book and sell your finished game to a software house, then you *must* have the author's written permission before you start. He owns *all* the rights in his book, including computer game rights!

This particular game is based on a sequence in which the central character searches through the nine worlds described by the Norse legends for a clue to help him defeat the evil forces that he has accidentally unleashed. In the novel it isn't entirely clear if this is 'really' happening or not, but the character *believes* it's real. His journey takes him from the Corpse Strand (where the game starts) up the high cliffs behind it to the roots of the World Tree, and then up inside the tree itself to find what he seeks among the high branches. At the end of the quest he finds the clue he needs, and is ready to return to his own world.

So how do you turn that into a game?

As a game writer, you have one or two wonderful luxuries that a novelist doesn't have. You aren't just writing a story with a beginning, a middle, and an end, a story that has to follow a single chosen track. You're creating a series of possibilities, a series of events that are more or less likely to happen according to the choices your player makes. The only things that are fixed about the game are the locations where the events happen, the other characters in the game, the objects available to

help or hinder the player, and the traps you set in his path. A really entertaining game never plays quite the same way twice – a good example is Legend's game *Valhalla*, which has so many random possibilities built into it that it remains endlessly fascinating and endlessly enjoyable. However, even those endless possibilities are based on a core idea; like any game, there is some sort of objective for the player, something he has to do. The great thing about many adventure games is that he can only discover the objective by playing the game, but the same shouldn't go for the game designer. Plot out your storyline before you start thinking about any details at all, and certainly before you get anywhere near the computer!

The storyline for *The Ice King*, as for many adventure games, depends on the player passing safely through a whole series of different locations, solving problems and making choices along the way. He may or may not be able to escape the consequences of making a wrong choice, and there will be several fatal traps lying in wait for him. Adventure games are *meant* to be frustrating! In effect, then, each location in the game is like a short chapter in a book, and each deserves as much care as you can give it.

On location

Locations in adventure games are often called 'rooms', because in most of the older games they *are* rooms. It's important to understand that a location can be any place, anywhere. The first location of *The Ice King* is a particular part of the Corpse Strand, and there are several other locations along the Strand, including two inside that mysterious dark hall.

The way you describe your locations is important. Your words, and your words alone, give the player his ideas about the game landscape. If I'd described the Corpse Strand as 'a beach under a grey sky' it would have sounded about as exciting as Brighton on a wet Sunday afternoon. Take trouble with these descriptions, and try to make them come to life. The players won't be able to guess what you had in mind when you wrote the game, it's up to you to tell them. Shut your eyes and try to imagine yourself at the location. What can you see as you look around you? Can you hear anything? Can you *smell* anything (no, I'm not kidding)? A few telling words, especially words that appeal to the senses, can make all the difference here. For instance:

A high branch of the World Tree. Far below, whirling stormclouds conceal the lands of men – the thunder so far away it sounds like distant drumbeats. Cloud castles hide many surrounding branches, and a creeping fog drifts about your ankles. Your hands are numb with bitter cold. Ahead, your path disappears into a mist-shrouded grove.

Feeling chilly yet? If you were actually playing *The Ice King*, and you'd just survived a dozen assorted supernatural terrors to get to that branch, you might feel

different, and that fog is obviously going to be a problem. Before long you'll be walking blind, and that's death in most adventure games!

The main reason for writing all this down at an early stage is to get a clear idea of what you are letting yourself in for at the programming stage. Adventure games have a habit of spreading themselves in a terrifyingly expansive way. Suddenly you have a game with 310 locations and 200 objects, and a computer that might just manage 48 locations and 20 objects! Thought at this stage will really pay dividends! Keep a careful tally of the objects you're placing, and a careful check on the real usefulness of each location *in game terms*. Sometimes you may be tempted to put in a location just because it seems a good idea at the time – you're proud of the description, perhaps, or there's a private joke in it that you can't quite resist. Resist it. Does anything here advance the player's search for his goal? Is there a trap here that might stop a greedy or over-confident player? Can you use this location to hide an object that the player needs? Can you use it to misdirect him, send him off in the wrong direction, or on a wild goose chase after something he doesn't actually need? If your location does none of these things, then forget it. Just as descriptions and dialogue in fiction are boring unless they also advance the story, so your location descriptions must have a place in the adventure as a whole, and that means knowing exactly where they are, and exactly how they relate to each other. The best way to do that is to draw a map.

Mapping out the game

Drawing a map for an adventure game isn't quite like drawing any other kind of map. For one thing, consistency is very important. If you go North from room 1 to room 2, then going South from room 2 should take you back to room 1 unless something very peculiar is happening to time and space. Yet it's surprising how many games are published with elementary mistakes like these uncorrected. If you think of each location as a cube, then there are six main directions in which you can leave it – North, South, East, West, up and down. However, many games also use Northeast, Southeast, Southwest and Northwest, giving a total of ten directions. How do you show this on a map?

My personal method is to use what I call 'separated squares' (see Fig. 16.1). Each square can have up to eight lines radiating outward from it: if there are exits from a square up to, say, location 19 and down to location 43 then you can simply write in U19 and D43 to key in to your maps of the upper and lower levels. Within the game framework two adjacent squares might actually be hundreds of miles apart – it doesn't matter, as long as you can keep track of the relevant entrances and exits. To do this you can draw lines linking the squares, and use arrowheads to show the direction in which motion is allowed. You might, for instance, be able to go out of a building to the Northeast, but find when you turn round to go back that the door has been locked against you. (Too bad if you forgot to pick up the key earlier on!)

Obviously all this is going to be pretty difficult if you don't already have a clear

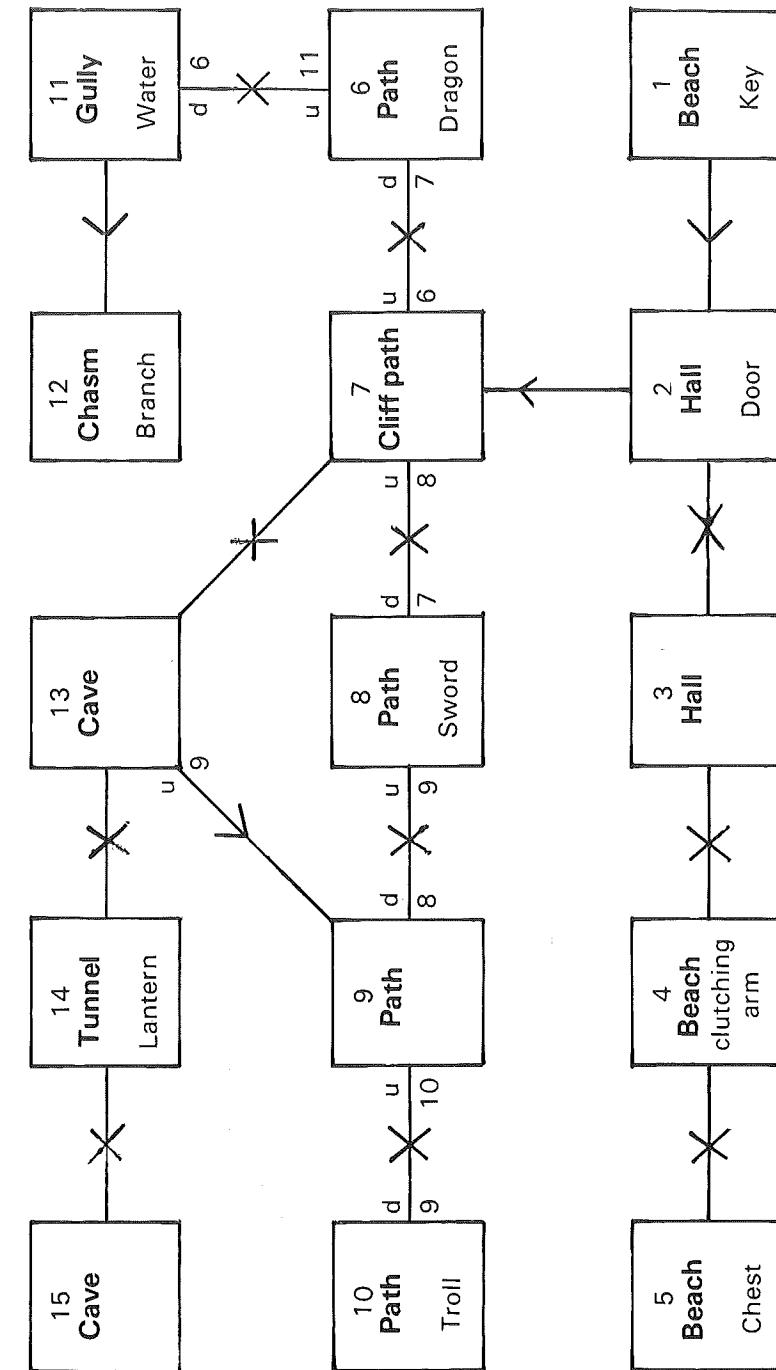


Fig. 16.1. A 'separated squares' adventure map. Professionals use octagons rather than squares, but they can be a bit hard to draw!

idea of your various main locations and what's in them. That's why I stressed the actual writing first! By now you should have a set of location descriptions drafted and ready to go, and the map should simply be a neat way of sorting out the final details. If it's more than that, you may be making trouble for yourself, and you'll almost certainly have to redraw the map several times, which could be very irritating. It's usually best to start in pencil, and ink everything in when you're quite sure it all checks out. This is especially useful for the first rough draft of an adventure that may be compressed a great deal before it's finally programmed into the computer.

Colour coding can be very useful at the mapping stage, particularly for sorting out the different exits and entrances from a room. No arrow from a wall means no exit that way, but a green arrow could mean an open door, a yellow arrow a door that only exists if certain conditions are met, and a red arrow could mean a locked door. Yet another colour might show that the door led to a location somewhere else on the map, rather than into the game room itself. How exactly you do this is up to you – work out a system that you find easy. However, if the squares become inconvenient (I use them because they're easy to draw!) you could try the more professional mapping method of drawing linked octagons, so that each of the eight major directions that might be used as exits or entrances to a room is marked by a corresponding line on the map. In big, complex adventures this could make your job a lot easier, but you're going to have to draw the octagons yourself!

Getting started

So much for the theory. Before you actually get down to writing your first adventure, here are a few things to think about.

First, don't try to be too ambitious, especially to start with. While you're getting the hang of writing adventure programs, stick to games with about twenty rooms and about as many objects. If you have other characters, keep them simple at this stage until you've mastered the programming involved. Think about the people who will be playing your game – remember that they don't know it as well as you do, and a problem that looks simple to you may be almost insoluble to someone else. If they don't know that they have to have both the key *and* the book before they can open that door in location 12, and you've hidden the key so cleverly they can't ever find it, then you may simply finish up with a very frustrated set of players. Be fair, within limits. You're expected to be a *little* sneaky! When in doubt, always do what you think will interest the player – if you want to sell games commercially, you must make them exciting and interesting *all* the time.

One more thought. If the entire object of the game is to achieve one goal, then the game is useless as soon as that object is achieved. Try to keep a random element in the game. For instance, arrange for there to be a number of possible goals, not always achieved in the same way, or a number of different routes that each lead to a different adventure. One quest might be for an object that will open the way to new locations – one of the most enjoyable features of *Valhalla*, for instance. With

care and thought, your game could give people a great deal of pleasure for a very long time.

And now for the programming ...

You've mapped out your game world, and written your location descriptions. You have checked that there's a safe way through to the final objective(s). You have a complete list of the objects available to the player within the game. And now you're ready to begin programming. And you may well be wondering how on earth to start. Actually, there are three options. One is to use one of the commercially available adventure game toolkit programs, which effectively does most of the hard work for you, and will also save you a great deal of programming, Second is to use a simple BASIC framework which is easy to understand and follow, but not as efficient in its use of memory as some. Third is to use more advanced programming techniques that will allow you to pack out your Spectrum's memory with room descriptions, objects, random monsters, and all the other programming paraphernalia of the typical adventure game. It's time now to look at all three methods in a little more detail.

Chapter Seventeen

Programmed for Adventure

Programming an adventure game sounds very complicated. After all, there are a great many details to take care of! The program must contain complete descriptions of every location and every object. It must be able to recognise valid movements from one location to another, and prevent unauthorised movements. It must be able to recognise and act upon every command that is supposed to be acceptable to it. It must keep track of what objects your character is carrying. It must be able to deal with the many special situations you want to include – for instance, arranging for interactions between the characters you control and other characters (or creatures!) controlled by the computer. There's no denying that all this can take a great deal of coding, even for a fairly small adventure. Fortunately, those of you who don't really fancy the idea can take a perfectly respectable and easy way out.

Introducing the instant adventure

Several of the adventure games now on sale were not programmed 'from the ground up' at all. Instead, they were prepared using an adventure game 'toolkit'. There are two of these programs available, one called *The Quill*, and another called *Dungeon Builder*. Programs like these take most of the hard work out of adventure game programming. You still have all the creative work to do, and a certain amount of careful structuring, but you do it painlessly and quickly using special routines devised to make it as easy for you as possible – and fairly economical on memory space, too. Both programs have their good and bad points, and it's worth taking a quick look at how they compare with each other.

The Dungeon Builder

This is a cheap, well designed and incredibly versatile program. Unfortunately, it's also very hard for a first time user to find his way around in it. *Dungeon Builder* works with a series of menu-driven routines that allow you to do everything necessary to create the adventure, including, for the very first time, a special routine for creating pictures of the locations. There's even a map within the program itself (a linked octagon design complete with colour codes for different kinds of door!) and special routines for creating monsters. *Dungeon Builder* is a professional adventure designer's dream, and was obviously intended to be – it will even accept and interpret multi-element commands such as GET AXE AND SWORD AND GO NORTH; this sort of command is much less limiting for the

player and helps to make his participation in the game more exciting and more involving. Another valuable feature is the weight assigned to individual objects. The built-in programming prevents a player carrying more than 50 kg weight, and you assign weights in tenths of a kilogram from 0.1 kg to 25.5 kg.

Not all the benefits of this remarkable program are obvious when you're using it. Most notable in this respect is the coding that 'compresses' room and other descriptions so that you can pack many more words into the game, words that may make the difference between yet another version of *Dungeons and Dragons* and a genuinely original contribution to the world of adventures. Believe it or not, you can do this yourself using simple BASIC – Chapter 18 will show you how to set about it.

The snag with *Dungeon Builder* is that it takes time and a certain amount of effort to learn its use in practice. You can program *something* almost at once, but making the program do exactly what you want it to do is a bit more difficult. If you think you're going to be serious about adventuring, I can heartily recommend it; it's just what you've been looking for. Almost my only complaint is the lack of a randomising facility, so useful in adventure games to ensure that they don't play the same way twice. The documentation is rather heavy weather, too, but I've already found that a certain amount of work with the program does increase your confidence and ability, so it's worth being patient. However, I do miss the user-defined variables offered by *The Quill*.

The Quill

The Quill is an excellently designed and documented program, but it's more expensive than *Dungeon Builder*, it doesn't allow you to create pictures, and it doesn't include a map in the software, so you'll have to draw your own. However, its remarkably clear structure makes a good model for adventure game design in BASIC, so it's worth seeing how *The Quill* works in practice.

After you've LOADED it from tape, *The Quill* presents you with a long menu listing all the various components of your adventure as follows:

The *vocabulary*. A list of all the words you want the computer to be able to recognise in commands.

The *message table*. Into this you program all the messages the computer will give in response to specific actions: for instance, if you try to do something impossible the computer could be programmed to PRINT the message 'Don't be silly!'

The *location table*. This is where you put your detailed location descriptions.

The *movement table*. This table is used to code the possible movements out of each room. For instance, the code N19 would mean that room (or location) 19 lies to the North of the location you're programming for.

The *object table*. This stores descriptions of all the portable objects used in the game.

The *object start location table*. This indicates which rooms the objects are in at the start of the game. Special codes are used to indicate objects that are worn, carried, or 'not created'. For instance, an open cupboard is 'not created' until the door of an existing closed cupboard has been opened!

The *event table* is one half of the program that actually operates the game. Its job is to translate commands from the keyboard into actions within the program. To use it, you key in the relevant command words (*The Quill* recognises only the first four letters of each word), followed by any conditions that need to be fulfilled, and then the commands the computer needs to carry out the action commanded. You can make several entries using the same word or pair of words (to deal, for instance, with similar situations in different locations), but the computer will deal with them in the order in which you entered them.

The *status table* is the other half of the operating program, corresponding to D & D's dungeon master – here you enter commands carried out by the computer in response to particular combinations of circumstances. For instance, if a player is in a particular location and carrying a particular object, you can arrange for the roof to fall on his head.

These are the main operating sections of *The Quill*, though several other very useful commands are also available from the menu. However, don't think that you can just dive in and start writing your adventure: working with *The Quill* takes just as much planning as working without it. For instance, if you try to use words in the event or status tables that haven't been programmed into the vocabulary you'll get an error message, and if you don't have a separate list of the message, object, and location texts then you can waste a great deal of time checking out the one you want. The program does include a facility for printing out these tables from the program's own arrays, but this is only designed to work with ZX or ZX-compatible printers, such as the Dean Alphacom.

Within the program itself, you also need to take a certain amount of care. The order in which entries are made in the status table can affect the outcome of your programming – *The Quill* deals with these entries in strict sequence. However, it isn't hard to learn, and before long you will be able to develop many useful tricks to expand the basic range of facilities the program gives you.

One particularly useful feature is a set of 33 variables, some of which are changed by normal program events such as accepting a command or describing a location, and some of which can be programmed to change using status or event table programming. *The Quill* arrives with a ready-made counter for the number of objects the player is carrying, but using the flags you can use more sophisticated systems. For instance, you could simulate the 'weight' variable system of *Dungeon Builder* by arranging for each object in the game to have a weight which was added to a variable when the object was picked up, and subtracted when it was dropped. That way if the player tries to carry a kitchen stove, a suit of armour, and a treasure chest you can tell him he's broken his back. (If he were allowed to carry three

objects, this absurd situation would pass unremarked!) Other variables might be used to indicate the player's own physical and mental characteristics within the game – more about this in a moment. The *Quill* manual also gives detailed instructions about light and dark locations – another useful trap! If a location is dark you can arrange for most commands to fail until the player finds a source of light!

Beating the limitations

At first sight, *The Quill* could seem rather limiting to experienced adventure game programmers. You can only use very simple low resolution graphics (or a few high resolution graphics using user-defined characters), and it will only accept two-word commands, usually in the form VERB NOUN (such as GET FISH, or DROP ELEPHANT). It can also accept one-word commands for movement (N, S, rather than GO NORTH, GO SOUTH). It can't handle multiple commands (GO NORTH AND GET CORKSCREW), and there is no way you can persuade it to do so. This said, there are few other limitations for the imaginative programmer: you just have to think! For instance, there is no obvious provision for other characters in the game – at least, not for other characters who behave in an independent and apparently 'intelligent' way – but I have managed to simulate it by treating such characters as a special type of object! Despite these problems, which are often more apparent than real, I think only a real programming purist would object to the use of *The Quill*! All the same, I'd like to spend a little time on a system of BASIC adventure programming for the Spectrum. It's not, obviously, as fast as *The Quill* in action, but it can be made quite fast enough for serious purposes; and although programs written this way may look complex when they're finished, the principles behind them are very simple.

Doing it yourself

The successful BASIC adventure program depends on a couple of fairly simple programming ideas already discussed in Part 1 – in other words, there's nothing new here! If you had visions of infinite complication, forget them: adventure game programming is very simple, there just tends to be rather a lot of it.

If you're still a little unsure of your BASIC programming, take a quick look back at Chapters 3 and 4, which deal with variables, strings, and arrays, and also with READ...DATA and FOR...NEXT loops. These are the prime tools of the adventure game programmer, and to help you understand them I'll take the various sections of *The Quill* described above and show how you can convert them into BASIC. To make this as easy as possible, the numbering in the program examples will actually work as part of a complete BASIC listing, so if you patiently work your way through this chapter, you'll finish up with a miniature adventure based on three locations in *The Ice King*. To demonstrate the principles here, I've rather overpacked this adventure. Normally you could get through at least two or three locations without encountering deathtraps!

(1) Vocabulary

The program must be able to translate commands typed in from the keyboard into something the computer can understand. This involves the apparently tricky business of recognising words in the first place. One simple system that doesn't involve too much memory space is to work with the first two letters of each word, and restrict keyboard commands to two words – VERB NOUN – as in *The Quill*.

Before it can recognise command words, the computer needs to know which words are valid. The simplest way to do this is to set up two arrays, one for verbs and one for nouns. Figure 17.1 shows how this can be done using READ...DATA loops, and in this case I've called the arrays v\$ (for Verb) and n\$ (for Noun). A DIMensioned array like this one is reasonably economical on memory space as long as it's fully packed, and you'll be using quite a number of other, similarly prepared arrays in the course of the program. To save memory space in a long adventure you can actually create these arrays in a completely separate program, as described in Part 1, and then copy them into your main program using commands like LOAD "verbs" DATA v\$(). It's important in adventure game programming to use memory as economically as possible!

```

997 REM ****
998 REM VOCABULARY ARRAY
999 REM ****
1000 DIM v$(11,2): FOR j=1 TO 11
: READ v$(j): NEXT j
1010 DATA "N ","S ","E ","W ","I
N","LO","GE","DR","US","OP","CL
"
1015 REM NORTH, SOUTH, EAST, WES
T, INVENTORY, LOOK, GET, DROP,
USE, OPEN, CLOSE
1020 DIM n$(7,2): FOR j=1 TO 7:
READ n$(j): NEXT j
1030 DATA "BO","SA","DA","KE","A
X","DO","CH"
1035 REM BOAT, SAIL, DAR, KEY, A
XE, DOOR, CHEST

```

Fig. 17.1. Vocabulary array using READ...DATA loop. Note that this is the first section of a long subroutine also including the message array, the location table, the movement, object, and object start location tables, and a routine to initialise program variables.

Once you have your arrays set up, you need a way of checking commands INPUT from the keyboard against them. Figure 17.2 is a program that selects the first two letters of each INPUT command word and checks them against the array. When it finds a match, it returns the number value of that word to the variables 'verb' and 'noun' in the main program. (You can use shorter variable names to save memory space – I've used these to make it clear what's going on.) Because each command *must*, according to the rule we've just set, begin with a verb, you can

```

1997 REM ****
1998 REM ENTER A COMMAND
1999 REM ****
2000 DIM a$(14): LET verb=0: LET
  noun=0: INPUT "What do you wan
t to do?": a$: PRINT ''a$
2010 FOR j=1 TO 11: IF a$( TO 2)
=v$(j) THEN LET verb=j: IF ver
b>6 THEN GO TO 2030
2015 IF verb>0 AND verb<=6 THEN
  RETURN
2020 NEXT j: RETURN
2030 FOR j=3 TO 14: IF a$(j)=" "
  THEN LET b$=a$(j+1 TO j+2): G
O TO 2050
2040 NEXT j: RETURN
2050 FOR j=1 TO 7: IF b$=n$(j) T
HEN LET noun=j: GO TO 2070
2060 NEXT j: RETURN
2070 RETURN

```

Fig. 17.2. Command entry subroutine. This simply assigns number values to 'verb' and 'noun' according to the command entered (capital letters *only* please!) If 'verb' lies between 0 and 6 then no noun is required and the RETURN command is executed.

then use the value of 'verb' to send the program to a subroutine where there is more programming to deal with the commands using that particular verb. These subroutines will be your equivalent of *The Quill's* event table, and we'll look at them more closely in a moment.

This system isn't perfect, and can be fooled – if your player enters a nonsense word that happens to start with the same two letters as a genuine command word, the computer can't be expected to tell the difference! Notice also that complex directions such as NORTHEAST *must* be abbreviated in use, because otherwise they'll be treated as simple directions (after all, the first two letters are NO, which the computer could read as NORTH). However, I can guarantee that this system will work most of the time, and certainly often enough to be useful!

(2) The message table

Again this is a string array, and again it can be programmed with a READ...DATA loop like the one in Fig. 17.3. The DATA array created here – "message" DATA m\$() – will effectively leave you with a list of numbered messages. For instance, a command to the computer to PRINT m\$(1) will lead to the message "OK – that's done" appearing on screen. One word of warning, though: try to keep your messages short to save memory space. If necessary, print several short messages one after another to make up a longer one. Remember that the array has to be DIMensioned – if you have space for 32 messages each 100 characters long, and most of your messages are about 15 characters long, then

```

1097 REM ****
1098 REM MESSAGE ARRAY
1099 REM ****
1100 DIM m$(3,19): FOR j=1 TO 3:
  READ m$(j): NEXT j
1110 DATA "OK - that's done.", "S
orry, you can't.", "You can't ca
rry it"

```

Fig. 17.3. A simple message array.

you've wasted a great deal of space! (The computer will automatically fill up all those empty spaces with blanks, and each of those blanks is taking up memory that you could use for something else.)

```

1197 REM ****
1198 REM LOCATION TABLE
1199 REM ****
1200 DIM r$(3,384): FOR j=1 TO 3
  READ r$(j): NEXT j
1210 DATA "You are on a beach be
neath a sunless sky on the b
rink of a vast ocean. Northwa
rds a black cliff reaches to the
clouds, featureless and uncl
imbable. Slow, oily waves was
h bleached white flotsam onto t
he beach. Scattered across the
beach are white, upstanding st
icks that seem to grow unmoved
by the windwhile westwards stan
ds a dark hall. You hear a mo
aning sound."
1220 DATA "You stand within the
dark hall. The sound of moaning
is much louder here – it is
the cry of hundreds of tormente
d souls that writhe in shadow, an
d behind it you can hear the thr
eatening hiss of something th
at might be a colossal serpent."
1230 DATA "You are on board a sm
all wooden boat, fitted with a s
hort mast. There is a skeleton
under the rowing bench, and wa
ter in the bilges."

```

Fig. 17.4. The location table. Notice how much space is wasted by the description in lines 1220 and 1230!

(3) The location table

As shown in Fig. 17.4, this can simply be a grander version of the message table, but it poses similar problems! If your location descriptions vary wildly in length, then you're going to waste a good deal of space. Another method is shown in the next chapter – it works by creating a single massive string from which you 'slice' the sections you need, a technique that can be used a great deal in adventure games.

(4) The movement table

Figure 17.5 shows a movement table for this mini-adventure, and it probably doesn't look like very much. However, the way it works is delightfully simple. The

```

1297 REM ****
1298 REM MOVEMENT TABLE
1299 REM ****
1300 DIM m(3,4): FOR j=1 TO 3: F
  OR k=1 TO 4: READ m(j,k): NEXT
  k: NEXT j
1310 DATA 0,0,0,2: REM Loc. 1
1320 DATA 0,0,1,0: REM Loc. 2
1330 DATA 0,3,3,3: REM Loc. 3

```

Fig. 17.5. A simple table for movement in four directions. Number arrays like this are only really economical of memory space if they hold *large* numbers.

movement table actually consists of a number array. Array m(1), for instance, gives all the different movements possible from room number 1. In this simple program only four directions (North, South, East and West) are used, but you could use many more. In the case of m(1) the array reads 0,0,0,2: in other words, the only exit is West, to room 2, and all other exits are blocked. Notice that the directions are listed in the same order here and in the vocabulary array – in other words v\$(3) is EAST (or EA, to be precise!) and m(1,3) gives the result of moving EAST from room 1. This sort of planning makes your programming much simpler!

(5) The object table

Sorry – yet another string array (see Fig. 17.6)! With this one take care to put the descriptions in precisely the same order as the words in the noun part of the vocabulary, even if you have to leave blanks sometimes. You shouldn't have to leave many.

(6) The object start location table

Again this is a simple number array (see Fig. 17.6), with one number for each object, and laid out in the same order as the noun and object tables. For instance, object 2 is the noun SAIL, which appears as SA in the noun array (n\$(2)), and is described in o\$(2), as 'A canvas sail'. Its number in the object start location table is

```

1397 REM ****
1398 REM OBJECT TABLE
1399 REM ****
1400 DIM o$(7,32): FOR j=1 TO 7:
  READ o$(j): NEXT j
1410 DATA "": REM Boat
1420 DATA "A canvas sail"
1430 DATA "A pair of oars"
1440 DATA "A rusty iron key"
1450 DATA "An iron-bound steel-bladed axe"
1460 DATA "": REM Door
1470 DATA "A heavy wooden chest"
1497 REM ****
1498 REM OBJECT START LOC. TABLE
1499 REM ****
1500 DIM o$(7): FOR j=1 TO 7: READ o$(j): NEXT j
1510 DATA 255,3,3,255,254,255,255

```

Fig. 17.6. The object and object start location tables.

3, indicating that you'll find it in location 3. I use a zero code in this table to mean an object that is carried (since there is no location zero), 254 to mean it's inside the chest (!) and 255 to mean that it's 'not created' – e.g. if it's not something you can actually pick up, or you haven't yet done the things you must if the object is to exist.

(7) The event table

As noted above, this is a series of subroutines reached by using the value of 'verb' returned from the vocabulary table. What you do in each subroutine depends entirely on what you want to happen in the game, so I can't lay down rules for it, but Fig. 17.7 shows you what *can* happen. These subroutines form the real heart of your program, so take trouble with them. Experiment, and keep experimenting until you get the results you want. You can increase the spacing between subroutines by altering the program line that calls them – some routines dealing with vital commands could demand a great deal of programming!

```

2997 REM ****
2998 REM EVENT TABLE
2999 REM ****
3000 IF verb<=4 AND verb>0 THEN
  GO SUB 3100: RETURN
3010 IF verb>6 AND noun=0 THEN
  LET message=2: RETURN
3020 GO SUB 3000+verb*50: RETURN

```

```

3100 IF loc=3 AND o(7)=255 THEN
  LET boat=1
3110 IF m(loc,verb)<>0 THEN LET
  loc=m(loc,verb): CLS : IF loc<
4 THEN PRINT r$(loc): RETURN
3120 IF loc=4 THEN CLS : GO TO
9500
3130 PRINT '"Sorry, you can't g
o that way"'; RETURN
3249 REM *****INVENTORY*****
3250 LET k=0: PRINT '"You have:
": FOR j=1 TO 7: IF o(j)=0 THE
N LET k=1: PRINT o$(j)
3260 NEXT j: IF k=0 THEN PRINT
"nothing at all"
3270 RETURN
3299 REM *****LOOK*****
3300 LET k=0: FOR j=1 TO 7: IF o
(j)=loc OR (chest=1 AND o(j)=25
4 AND (o(7)=loc OR o(7)=0)) THEN
  PRINT o$(j): LET k=1
3310 NEXT j: IF k=1 THEN PRINT
"can be seen"
3320 RETURN
3349 REM *****GET*****
3350 IF inventory=2 THEN LET me
ssage=3: RETURN
3360 FOR j=1 TO 7: IF o(noun)=1o
c OR (chest=1 AND (o(7)=loc OR
o(7)=0) AND o(noun)=254) AND in
ventory<2 THEN LET o(noun)=0: L
ET inventory=inventory +1: LET m
essage=1: RETURN
3370 NEXT j: LET message=2: RETU
RN
3399 REM *****DROP*****
3400 IF chest=1 AND o(noun)=0 AN
D noun<>7 THEN LET o(noun)=254
: LET inventory=inventory-1: LET
message=1: RETURN
3410 IF o(noun)=0 THEN LET o(no
un)=loc: LET inventory=inventor
y-1: LET message=1: RETURN
3420 LET message=2: RETURN
3449 REM *****USE*****

```

```

3450 IF noun<4 AND o(noun)=0 AND
loc=3 THEN LET boat=1: LET o(
noun)=255: LET inventory=invento
ry-1: LET message=1: RETURN
3460 IF noun=5 AND o(noun)=0 AND
loc=3 THEN PRINT '"Oops! You
've just smashed the planking
on the port side. Water is pourin
g in....": GO TO 9000
3470 IF noun=4 AND chest=0 THEN
  LET noun=7: GO SUB 3500: RETUR
N
3480 IF noun=5 AND loc=2 AND doo
r=0 THEN LET door=1: LET m(2,1
)=4: LET message=1: RETURN
3490 LET message=2: RETURN
3499 REM *****OPEN*****
3500 IF noun=6 AND loc=2 THEN P
RINT '"What with?": RETURN
3510 IF noun=7 AND chest=0 AND (
o(7)=loc OR o(7)=0) AND o(4)=0
THEN LET o$(7)="An open chest":
  LET chest=1: LET message=1: RET
URN
3530 LET message=2: RETURN
3549 REM *****CLOSE*****
3550 IF o(4)=0 AND chest=1 AND (
o(7)=0 OR o(7)=loc) THEN LET c
hest=0: LET o$(7)="A heavy woode
n chest": LET message=1: RETURN
3560 LET message=2: RETURN

```

Fig. 17.7. The event table – the heart of the program. Notice how it consists of a series of subroutines keyed to individual verbs in the vocabulary array.

(8) The status table

The program will check this subroutine every time it executes a command, so to save time I generally draw it up as a series of subroutines each keyed to a particular location number (see Fig. 17.8). However, you can also include a short routine to check on the player's 'vital statistics' – strength, hunger, etc. For instance, if the player's strength is held by the program as a variable that decreases by one every time a command is entered, you could easily create a routine to warn the player when he needs food. In a working program you'd probably include the warning as one of the messages in the message table.

```

3997 REM *****
3998 REM STATUS TABLE
3999 REM *****
4000 GO SUB 4000+loc*100: RETURN

4099 REM *****Location 1*****
4100 IF o(7)<>255 AND key=0 THEN
    LET o(4)=loc: LET key=i
4110 IF verb=6 AND beach=0 THEN
    LET m(1,2)=3: LET beach=1: PRINT
    "The sticks in the sand are
    human arm-bones, and the
    beach itself is formed from the
    crushed remains of millions of
    bones."
4120 IF verb=6 THEN PRINT "You
    can see a boat drawn up on the
    beach to the south."
4130 RETURN

4199 REM *****Location 2*****
4200 LET end=end+1: IF end=4 THEN
    PRINT "The hissing rises to
    a deafening pitch and you are seized
    and enwrapped by dozens of
    venomous serpents...": GO TO 9000
4210 IF verb=6 AND (o(5)=0 OR o(
    4)=0) THEN PRINT "You can see
    a shadowy door to the north.
    ": RETURN
4220 IF verb=6 THEN PRINT "You
    can only see dark shapes moving
    in the shadows.": RETURN
4230 IF verb=1 AND door=1 THEN
    LET m(2,1)=4: RETURN
4240 RETURN

4299 REM *****Location 3*****
4300 IF verb=6 AND o(2)<>loc AND
    o(7)=255 THEN PRINT "Now that
    you've moved the sail you can
    see a large wooden chest next to
    the skeleton.": LET o(7)=loc:
    RETURN
4310 IF o(7)<>255 AND boat=0 THEN
    LET m(loc,1)=1
4320 IF boat=1 THEN LET end=end

```

```

4330 IF boat=1 AND monster=0 AND
    o(5)=0 AND RND*10<5 THEN PRIN
    T ""A ghoulish creature is climbing
    over the side of your boat.

```

```

    He looks very, very hungry...
    : LET monster=1: RETURN
4340 IF monster=1 THEN PRINT ""
    "You have fallen victim to the
    ghoul...": GO TO 9000
4350 IF end=6 THEN PRINT "You are
    dying... "
4360 IF end=7 THEN PRINT ""You
    are very near the end...": PAU
    SE 200: GO TO 9000
4370 RETURN

```

Fig. 17.8. The status table. Here a variety of death-traps are keyed in to individual locations in the game. Extra lines at 4000 could include more general things such as strength, courage, etc.

Practical programming

Whether you're 'rolling your own' program or using a toolkit program like *The Quill*, you must test your programming at every stage. With *The Quill* this is fairly easy, and special facilities are provided to help you, including a full screen display of the 33 variables available. With a BASIC program it's obviously a little harder, and you must make sure that *you* understand what the program is meant to be doing in each line. The listing in Fig. 17.9, used in conjunction with the other listings given above, will produce a miniature three-room adventure that should give you some idea of how these programs can actually operate. From then on, it's up to you! However, to help you along, the final short chapter of this section is devoted to refinements and extras that can help turn your adventure into something a little bit special. It may also give you some ideas for more advanced programming, especially ways to avoid wasting memory space in string arrays.

```

7 REM *****
8 REM MAIN OPERATING PROGRAM
9 REM *****
10 POKE 23658,8: LET loc=1: PRINT
    "You are lost near high cliffs, frozen and blinded by a violent
    snowstorm. You stumble, fall, and pass out. You wake in a
    strange and frightening place.": GO SUB 1000: PRINT r$(1
    loc): REM Fill arrays, describe Location 1.

```

```

20 LET message=0: GO SUB 2000:
REM Get command
30 IF verb<>0 THEN GO SUB 300
0: IF message<>0 THEN PRINT ''
m$(message): REM Check event tab
le
40 GO SUB 4000: REM Check stat
us table
50 IF verb=0 THEN PRINT ''m$(
2)
60 GO TO 20

```

```

1597 REM ****
1598 REM INITIALISE VARIABLES
1599 REM ****
1600 LET beach=0: LET end=0: LET
inventory=0: LET key=0: LET ch
est=0: LET door=0: LET boat=0: L
ET monster=0
1990 RETURN

```

```

9000 PRINT '''You have joined th
e legions of the undead in Hel
.'''Your bones will be washed o
nto the corpse strand and crush
ed into white shingle by the
ceaseless waves.'''Your sp
irit will haunt the dreadfu
l halls of Hel for ever.'': STOP
9500 PRINT '' FLASH 1;"Well done
!"
9510 PRINT ''Your skill and cou
rage have brought you safel
y through the perils of Hel. Yo
u have become a "; FLASH 1;"fir
st-grade warrior."; FLASH 0
9520 PRINT ''A steep path is no
w visible up the cliffs to the
north. Will you have the cour
age to program what happens next
...?": STOP
9998 STOP

```

Fig. 17.9. The three final routines needed to complete the mini-adventure. Line 9000 is particularly final ...

Chapter Eighteen

Finishing Touches

In the previous chapter you have seen a working adventure program, complete with death-traps, trickeries, hidden objects, random monsters and a nice, rewarding final frame (have you got there yet, or do you keep getting killed?) However, I certainly can't pretend that even a tightly packed three-room adventure is going to show you all there is to know about adventure game programming. This chapter is designed to give you some more ideas to try out for yourself; the routines within it are *not* designed to be used with the game in the previous chapter, so please don't try to MERGE them with it or you'll ruin a great deal of work!

Packing it in

The main problem with string arrays, as we have already seen, is that you can easily waste large chunks of memory. Because each array must be DIMensioned, every string in it must be as long as the longest string, even if that means leaving several hundred blank spaces. The program in Fig. 18.1 shows a different approach. Here all the 'strings' you need actually form part of a single giant string, and the LEN command is used to create a sort of index, which is stored in a number array. Number arrays, too, take up space – each item in such an array uses five bytes – but then each character in a string uses one byte, so you could easily save the number of bytes in the array ten times over using this program. (Incidentally, if you're not sure what a 'byte' is then take a quick look at the beginning of Part 5.) A program along these lines could be used for all the string arrays in your adventure, meaning that 'messages' could range from a couple of words to half a screen of text without wasting any more memory space than absolutely necessary.

It is actually possible to pack strings even more thoroughly than this, by arranging for single-byte codes to represent common groups of letters, but there isn't really room to deal with the necessary techniques in a book that has a fair amount of other ground to cover! However, the short routine in Fig. 18.2 should give you a start. The trick is that, as we have seen, all letters, numbers and punctuation marks are handled by codes. These have numbers from 0 to 127; but there are 127 additional codes available, used by the Spectrum for things like command words and graphics symbols. Within a program, you can make them stand for groups of letters of any length, or even for entire words, so saving quite phenomenal amounts of memory. (This is how Level 9 can pack games like

```

10 REM String Packer
20 PRINT "This program allows
you to build a single massive st
ring that can be used like an arr
ay."
30 INPUT "How many entries do
you want? ";n
40 PRINT '''If you ENTER somet
hing by mistake, you can
erase it by keying in X and E
ENTER.'''
50 LET a$="": DIM a(n+1)
60 FOR j=1 TO n
70 INPUT "Entry number "+STR$(
j+" is?";i$
80 IF i$="X" OR i$="x" THEN L
ET a$=a$( TO a(j-1)-1): LET j=j
-1: GO TO 70
90 IF a$="" THEN LET a(j)=1:
LET a$=i$: GO TO 110
100 LET a(j)=LEN a$+1: LET a$=a
$+i$
110 NEXT j: LET a(n+1)=LEN a$+1
120 PRINT '''Final entry now re
corded. If your last entry w
as wrong, key X and ENTER to er
ase it. Any other key to cont
inue.'''
130 INPUT i$: IF i$="X" OR i$="
x" THEN GO TO 80
140 CLS
200 REM String finder
210 INPUT "Entry number require
d? ";k: IF k>n THEN GO TO 210
220 PRINT a$(a(k) TO a(k+1)-1):
GO TO 210

```

Fig. 18.1. String packing program. It effectively turns a giant string into a pseudo-array using a number array and the LEN command.

Snowball into 48K!) If you're interested, why not try it and see how you get on?

Continued next episode ...

Another way to extend an adventure program is to create it in several parts. A

```

5 REM String Analyser
10 LET a$="BOB THE HOBBIT"
20 DIM b$(12,2): LET b$(1)="OB
"
25 LET c$=""
30 FOR j=1 TO LEN a$
35 IF j=LEN a$ THEN GO TO 50
40 IF a$(j TO j+1)=b$(1) THEN
LET c$=c$+CHR$ 128: LET j=j+1:
GO TO 60
50 LET c$=c$+a$(j)
60 NEXT j
70 PRINT "The first string was
";LEN a$;" characters long"'''T
he encoded one was ";LEN c$;"cha
racters long."
80 REM String Decoder
85 PRINT '''The original strin
g was:"'
90 FOR j=1 TO LEN c$
100 IF CODE c$(j)>127 THEN PRI
NT b$(CODE c$(j)-127);: GO TO 1
20
110 PRINT c$(j);
120 NEXT j

```

Fig. 18.2. Packing even tighter! This program shows how CHR\$ codes can be used to stand for letter groups in a string, and how to decode them during a game.

number of commercial games do this, and you can easily do it too. If you are careful, and cunning, you can even arrange for large chunks of your programming to apply to all parts of the adventure! For instance, your verb array could stay the same throughout, while the noun and object arrays change to suit a new set of objects available in the second part of the game. Be careful, though – is your character going to be carrying objects through from one part to the next? And if you have any other 'computer controlled' characters, can they cross the gap as well? Talking of characters, what about your hero(ine)?

Adventures of character

When I'm writing an adventure I always like to involve the player directly in the game with messages like 'You dissolve into a puddle of green slime' or 'You discover the lost hoard of the Incas and become disgustingly, mind-bendingly wealthy', but there are those who prefer to have a character within the game doing all the work. (GET GUN – Coolly Marlowe walks across to the desk and takes the

gun.) For the novelists among you this may be rather tempting, but I think the majority of players prefer a closer involvement. Certainly this has always been the main attraction of the original adventure games like *Dungeons and Dragons*, but the other attraction is that players can, to some extent at least, choose what sort of characters they want to be within the framework of the game.

In D & D this is done by throwing dice to achieve scores out of 20 for characteristics such as strength, intelligence, wisdom, dexterity, charisma, and magical ability. But as you already have a random number generator built into your computer, you can quite easily do this for yourself with commands such as

LET strength = RND*20+1

which will give you strength anywhere from 1 to 20 inclusive. This system is impartial, but not necessarily fair – you could end up with mostly low scores. It would be kinder, perhaps, to give the player a chance to choose his good points for himself: for instance, by giving him 80 points to allocate as he pleases among his six characteristics. Figure 18.3 is a short program showing how this might be done.

Having done all that, though, how do you use the figures you've got? In practice, the main use for them is often in what we've called the status table. For instance, you can arrange for the strength of your character to decrease a little every time a command is entered, and a little more if he is carrying heavy objects. To rebuild his strength he must find food, or drink, or some suitably reviving magic. Using helpful inscriptions requires intelligence, so you might arrange for the likelihood of finding them to depend on intelligence. Similarly, you can arrange for programmed encounters with other inhabitants of the game world to depend on these characteristics. A meeting with a powerful wizard may have a better chance of success if your character has magical ability, or unusual wisdom, or unusual intelligence. A meeting with a warrior will depend on strength and agility, or you might outwit him with intelligence. Figure 18.4 shows one such programmed encounter, and you can see that a strong element of chance is still involved.

Just as some abilities decrease with time, so others increase. If a character stays alive in the game world, he must have a reasonable level of intelligence, so you can increase his intelligence accordingly. As he grows 'older', his wisdom and magic skills will also increase, and if he passes successfully to a new level then his experience should serve him well. In D & D a system of 'experience points' is used – successful completion of a level increases a character's store of these, and at the end of the game they may increase his status, moving him up to the next 'level', where he will be capable of meeting even more severe tests and even tougher opponents. At the end of the mini-adventure you may notice that I laid the groundwork for something like this by describing the character as a 'first-level warrior'. For the player this has advantages – weaker opponents are easily disposed of – but the programmer should make sure that the character's greater powers also bring him up against bigger and better opponents! This isn't difficult to do, you can simply make the existence of such creatures in the game dependent on the character's abilities being well enough developed to have a chance of coping with them, for instance with lines like:

```

10 DIM a$(6,10): DIM c(6): LET
total=0
20 DATA "Strength", "Agility", "Bravery", "Knowledge", "Wisdom", "Charisma"
25 FOR j=1 TO 6: READ a$(j): N
EXT j
30 PRINT "CHARACTER BUILDER"
40 PRINT "' 40 points are available to you, to be divided among the six characteristics listed here. Each characteristic must have atleast 1 point, but none may have more than 10.'"
50 GO SUB 100: FOR j=1 TO 6
60 INPUT "Enter your "+a$(j)+" rating.", a: IF a=INT a AND a>0
AND a<11 THEN GO TO 80
70 GO TO 60
80 IF (40-(total+a))<(6-j) THE
N PRINT #0;"You only have ";40
-total;" points left!": PAUSE 15
0: GO TO 60
90 LET total=total+a: LET c(j)
=a: GO SUB 100: NEXT j: PRINT
"'Your ratings are displayed in
the table above. Good luck!'"
95 STOP
100 RESTORE : PRINT AT 10,0; "
: FOR k=1 TO 6: PRINT a$(k);"
":c(k),: NEXT k
110 RETURN

```

Fig. 18.3. Character building! This program allows the player to choose his own strengths and weaknesses at the start of an adventure.

```

20 PRINT "A shambling troll is
blocking the corridor. With
out warning, it swings a massive
club. To hit back, press any
number key from 1 to 9 – but the
harder you hit, the more strength
you use up..."
25 REM s=strength, a=agility,
m=monster

```

```

30 LET s=20: LET a=20: LET ms=
s: LET ma=a
40 PRINT AT 10,0;"Your strengt
h","Its strength"
50 PRINT AT 11,0;s:" ",ms:" "
55 IF s<0 OR ms<0 THEN STOP
60 LET mhit=INT (RND*9): LET m
s=ms-mhit
70 IF ma*RND>5 THEN LET s=s-I
NT (mhit/(ms+1))
80 LET h$=INKEY$: IF INKEY$="""
THEN GO TO 80
90 PAUSE 50: IF CODE h$<49 OR
CODE h$>57 THEN GO TO 80
100 LET hit=VAL h$: LET s=s-hit
110 IF a*RND>5 THEN LET ms=ms-
INT (hit/(s+1))
120 IF ms<0 THEN PRINT ' "We
l fought - you've killed it."
130 IF s<=0 THEN PRINT ' "Hard
luck - you've just joined the
choir celestial..."'
140 GO TO 50

```

Fig. 18.4. Combat with a troll. Strength and agility are set at the same levels for player and monster here, but there is a strong element of luck (do you hit him?) and intelligent play can easily defeat the random strikes of the computer controlled opponent.

5035 IF experience >20 THEN LET dragon=1

Because characters genuinely change in the course of a game, it may be useful to give your player the chance to 'rearrange' his character. One way is to arrange for some of those experience points to be redistributed among the characteristics, making the character stronger, more intelligent, or whatever. If you wanted to add more points to the charisma score you could finish up with someone irresistible to the opposite sex, or someone who could talk anyone into anything, but beware of allowing superpowers of any description. Characters should remain human, with human weaknesses, or they rapidly become very boring indeed. (After all, if SuperJane can do everything, where's the risk of 'death' that gives the edge to all adventures?) It's usually more useful to offer another short program section like the one in Fig. 18.3 at the end of each game section. Some of the points might even be used to let the character buy better equipment, such as armour and weapons.

Is there anybody there?

Talking of characters brings me to another important point – *game* characters. These shadowy creatures only exist within the boundaries of your game, so take

a little trouble with them, especially if the player is likely to have extensive dealings with them. 'You see a wizard' is boring. 'You see a tall, thin-faced man wearing a white robe' excites curiosity, and your player may not realise he's dealing with a wizard until the first spell is cast and battle is joined!

In the mini-adventure I deliberately didn't describe the 'ghoulish figure', mainly because I wanted to persuade excitable players to go for him with the axe, if they had it, and so have the exquisite pleasure of sinking their own boat. Actually you can't escape him, because he only pops up in a section of the game where you've already lost without knowing it! This is fair play, and sometimes allows your players to depart in a satisfying blaze of glory! But 'random monsters' are a regular and exciting part of D & D, and can be a useful addition to any game. Just remember to play fair and keep them weak enough for the player to deal with them under normal circumstances. A single line in the status table will actually be enough to set up this sort of situation, with the strength of the monster keyed to the player's own strength, e.g.

```

4000 IF RND*10+1<5 THEN LET monster = 1
4010 PRINT " "You see an oozing mound of green slime moving down the
corridor towards you."
4020 LET mstrength = INT (strength*.75)

```

In this case 'mstrength' is the monster's strength rating, which in this case is only three-quarters of the character's strength. The player, however, doesn't know this, and may well choose to run away. (Tidy minded players may object to the idea of cleaning green slime off their weapons.) Another useful gimmick is to describe a creature that is actually friendly in such a way that aggressive players immediately reach for their guns/swords/wooden sticks/laser tubes. As they contentedly chop the source of some vital clue into mincemeat you can sit back smug in the knowledge that indiscriminate violence is its own punishment. For instance, one of my own recent efforts involves a butler who follows the main character relentlessly around a huge house. If you hit him, he renders you unconscious, and you lose a fairly substantial number of strength points. If, however, you behave like a civilised human being and *talk* to him, he will ask politely for your hat and coat (which, if you are very careless, you may not even know you are wearing). The poor fellow has been following you all this while out of mere courtesy and hospitality.

Any old iron ...

Any adventure game, even the mini-adventure, is (and should be) littered with interesting objects. I'm not suggesting you pack them quite as densely as I packed mine – after all, I only had three locations – but do be imaginative. Some objects can be useless. Some can be essential. Some can be actively dangerous, in subtle ways (a harmless tipple early in the game may lead to a car crash later on, for instance). And some, of course, can be lethal! Once again, be fair. No harm in penalising the greedy or unthinking player, but make sure there is at least one safe

way through the maze – in your enthusiasm you may have destroyed the player's sole chance of survival with your last, admittedly brilliant, piece of programming. Here, once again, the value of planning everything beforehand becomes very clear, and once you *have* planned it, resist the temptation to improve it.

Finding and using objects is part of the fun of an adventure. Finding valuable objects and clues usually increases your score, or your experience points. Finding valueless ones can sometimes provide both programmer and player with some entertaining moments (after all, you want him to enjoy himself even if he's losing!). Imagine, for instance, a scenario in which one room contains a bottle opener, another some distance away contains a small and apparently useless key, and a third contains a rather dingy cupboard. However, use the key to open the cupboard and you discover a glass and a bottle of vintage wine. You will, however, need the bottle opener to get at it, and if you are then incautious enough to enter DRINK WINE a stern message will rebuke you for drinking fine wine out of a bottle. As for DROP GLASS – this produces the answer 'You've smashed it, you d****d hooligan!' None of these objects need be useful, they, and the messages they produce, are simply there for entertainment value. Do this too often and you'll run out of computer, but the occasional excursion into harmless humour is well worth it.

Traps and trickery

In the course of this and the previous chapter we've already seen a wide variety of traps for the unwary (and even for the wary). Again, use your imagination and be fair. In the mini-adventure, had I followed the book, anyone on the beach would have run the risk of being grabbed and torn apart by skeletal hands growing out of the sand. I resisted this because I felt the player needed at least one location where he could be safe to sit and think about what to do next. A subtler trap is the boat itself – try USEing the oars or the sail, and see what happens. In a larger game, especially, this might seem like a logical move, a way of escaping a difficult situation, and punishing such escape attempts is entirely fair! If a player wants to avoid problems, he shouldn't be playing the game in the first place.

Traps can be insidious, too. An apparently useful object may be so heavy that it rapidly saps your strength, so rapidly, in fact, that you can't eat or drink enough to function normally. The player who is intelligent enough to spot the problem and drop the object deserves something added to his score and a rise in intelligence points! In fact, traps like this can actually work to the player's advantage if he is sharp enough to see what's happening.

Beware of fatal traps. Too many, and your player will become frustrated; too few, and he may become bored. What keeps bringing me back again and again to some games is the sheer irritation I feel with myself for not being able to get past room X without being eaten, or rendered into cutlets, or transformed into a toad. However, if I can't even reach the second room without escaping some fiendish and totally unfathomable programming ploy I am liable to throw the game tape in the waste bin and watch TV instead.

Postscript

If you have just battled your way through untold horrors, and killed the master magician in the high tower of his great fortress after a battle that has shaken the entire planet on its axis, then you don't want to see a pitiful little message saying something like 'You have defeated the magician. Well done.' Ideally you want some flashing lights, some nice colours, and a multiple fanfare, so for your closing sequence take a good, hard look at Part 2. Similarly, a player who has managed to get himself killed deserves a little more than the stark 'You are dead.' I enjoyed writing the little end sequence in the last chapter (I hope you found it suitably creepy), and you can enjoy similar amusements. Take trouble over it – one of the best adventure-cum-arcade games I've ever played, *The Alchemist*, is seriously let down by a strangely unimaginative and anticlimactic ending that leaves you wondering why you ever bothered.

I hope this short section has given you plenty of ideas of your own. Please don't feel you have to restrict your adventure gaming to the sort of fantasy worlds I've often been describing here. Adventure games can be used, as I've said, for anything from sex education to science fiction, and most stops in between. Enjoy them, and enjoy programming them. There are few things that give more satisfaction than a successful and well written adventure program, and programs like that will give just as much satisfaction to the player.

Part 5

Extending the Spectrum

by Allan Scott and Ian Sinclair

Extending the Spectrum

It is perfectly possible that you find the idea of add-ons rather puzzling. Why, after all, should you want to add anything on to a computer that already has such an enviable reputation for features and memory size? Who needs add-ons?

You do. In fact you are using at least two of them already: your TV set and your cassette recorder are both add-ons (the proper term, incidentally, is *peripherals*) and without these peripherals you'd be pretty helpless. The computer doesn't need the TV set, it will function quite happily without it, but *you* won't, and that's the main reason for buying any peripheral. A useful peripheral is one that helps you do a particular task more easily. If it doesn't do that then you're wasting your money. As to the sort of tasks that peripherals can be used for – well, you're looking at one.

Ladies and gentlemen, a demonstration ...

This section was typed using a ZX Spectrum fitted with a **Fuller FDS** keyboard and a **Cub** monitor and running the excellent **Tasword 2** word processing program. The completed files were stored on Microdrive cartridges – when I'd finished with them I used a **Prism VTX 5000** modem and **OEL**'s new user-to-user software to send their contents down the phone line to Phil Gardner for copy editing. (He lives 300 miles away in Leeds, and sometimes it feels further than that.) At the other end, Phil was able to copy edit my work on *his* Spectrum and then print out the results on his **Lucas** Printer using a **Hilderbay** interface. Meanwhile, much of the artwork for the section was being prepared on my Spectrum using the **RD Digital Tracer**, and I was printing out the results, and a copy of the draft manuscript, using the **Morex** interface and a **Kaga Taxan** printer. Effectively, this whole section is a product of Spectrum add-ons, and if you don't feel ready to write your first book at the moment, then don't worry. That certainly isn't all the Spectrum can do.

If you have owned a Spectrum for some time then you probably know quite a lot about its capabilities, and you may already know how much more is possible. In the chapters that follow you'll find a great deal of information about all those tempting peripherals that appear so regularly in the computer press. If you're a newcomer to computing then this section will introduce you to a Spectrum you may never have known existed. It can become a high speed games machine. It can produce high quality printing on anything from a personal letter to a full length novel, using a comfortable custom-designed keyboard. It can access hundreds of

kilobytes of memory, in seconds (if you're not sure what kilobytes are, take a quick look at Chapter 19). It can talk to you, sing to you, play three-part harmonies for you, and help you to compose the next number one hit single. It can be used to copy photographs, drawings, original art, and your own screen designs. It can access huge banks of data on mainframe computers for little more than the cost of a phone call, and let you 'talk' to almost any other computer. It can even control household equipment – anything from calling you up at the office to tell you the house is on fire to operating a domestic robot.

Like any good computer, the Spectrum is a sort of ideal workman. If you give it the right tools and the right instructions, it will do almost anything. Why? That's a question I'll try to answer in the next chapter, and then we'll take a detailed look at the tools that are available, and the jobs they can do. Before that, a few vital words about fitting add-on units. Please read this section carefully: it could save you a lot of money, a lot of time, and a great deal of fruitless argument with manufacturers!

- *Always* make quite certain that power to your Spectrum has been switched off at the mains before connecting or disconnecting any add-on unit. Failure to do so could damage both the computer and the add-on.
- Before plugging in an add-on unit check that the *key* (usually a small piece of metal) is in place on the edge connector (see Fig. 19.10). If the key has fallen out and a connection is made, damage could result.
- When connecting equipment of any kind to the user port at the back of the Spectrum, make sure that the key lines up with the slot visible on the printed circuit board inside the user port (see Fig. 19.10).
- Some of the equipment mentioned in this book was originally designed for use with the ZX81. Edge connectors on equipment of this type are noticeably shorter than the matching circuit board in the user port, and you should take particular care when lining them up for connection. Check when first setting up equipment of this type that no converter circuitry is needed. Some units are supplied with extra connecting boards so they can be fitted to the Spectrum.
- Many add-on units have no rear connector for the addition of further units – they have to be 'the last in the chain'. You can beat this problem (at extra cost) by using cables and connectors available from good computer shops. This also solves the problem of linking up units with ZX81 and Spectrum connections, but it's expensive, untidy, and inconvenient.
- Don't overload the back of your Spectrum with add-on units. It is quite possible that one or more of them could be competing for the same area of memory, and if this happens the computer may well 'lock up': the screen will go blank, or a fixed pattern will appear, and no key will operate. If this happens to you, turn off the power *at the mains* (not at the back of the computer), disconnect one add-on unit at a time, and try to find which two units are in conflict. Incidentally, a lock-up only *looks* serious – all the same, it does mean you will lose any data that was in the computer at the time.

This section does not include detailed reviews of all the different add-ons available. That would take up a book by itself. However, Appendix C does give the names and addresses of the main add-on manufacturers and a brief listing of the products they can supply.

Chapter Nineteen

The Spectrum at Work

What exactly is a computer?

No, it isn't a trick question. The point is that you won't find it very easy to understand what the hardware in the next few chapters is and does unless you have a clear idea of what you're plugging it into. So even if you're sure you know exactly what a computer is, bear with the rest of us for a paragraph or two. We're going to take a look at the way it works in some detail, and, incidentally, give you most of the groundwork you need to understand the material in Part 6, the machine code section. If that puts you off, don't let it – the surprising thing about machine code is its simplicity.

In the beginning, then, there is memory. A unit of memory, as far as we are concerned, is just an electrical circuit that acts like a switch. You walk into a room, switch a light on, and you never think of it as remarkable that the light stays on until you switch it off. You never tell your friends, in reverent tones, that the light circuit contains a memory, and yet each memory unit of a computer is no more than a very small variety of switch which can be turned on or off and which will then stay that way until it is used again. One unit of memory like this is called a *bit* – the name is a contraction of *binary digit*.

Now let's stick with the idea of a switch, because it is so useful. Suppose that we wanted to signal with electrical circuits and switches. We could use a circuit like the one in Fig. 19.1. When the switch is on, the light is also on, and we take this as meaning YES. Turn the switch off, and the light goes out; we could take this to mean NO. You could use any other two meanings that you wanted, so long as there are only two. Things improve if you use two switches, two lights, two lines, as in Fig. 19.2. Now four different combinations are possible: (a) both off (b) A on, B off (c) B on, A off, or (d) both on. This means that we could signal four different meanings. Using one line gives two possible meanings; using two lines gives four meanings ($2 \times 2 = 4$), and if you feel inclined to work them all out you will find that using three lines will permit eight different combinations of signals and therefore eight different meanings. Since 8 is $2 \times 2 \times 2$, it should not be a surprise to learn that eight lines would allow you $2 \times 2 = 256$ different meanings to be communicated by means of signals. For N lines, the number of possible signals is 2^N , in fact. Any collection of eight switches, each of which can be on or off can be set into 256 different arrangements. It's up to us to make some sense of how we use these signals.

One particularly useful way is called binary code. Binary code is a way of writing numbers using only two digits, 0 and 1. The zero is often shown crossed to avoid

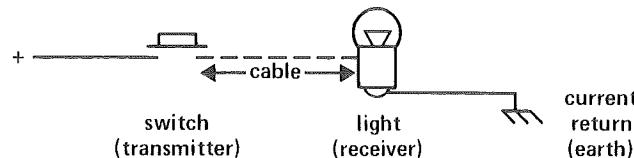


Fig. 19.1. - A single-line switch and bulb signalling system

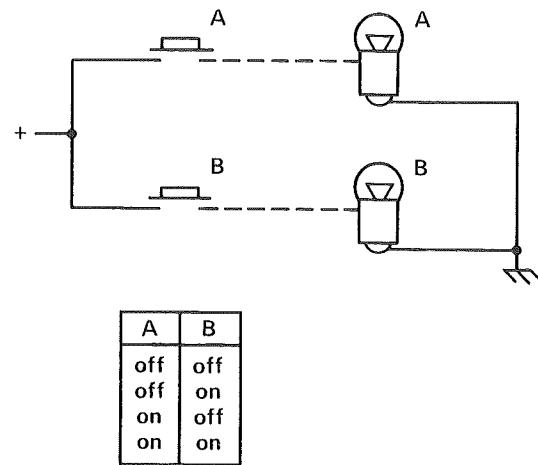


Fig. 19.2. Two-line signalling – four possible signals can now be sent.

confusion with the letter O. We can think of zero as meaning 'switch off', and 1 as meaning 'switch on', so that 256 different numbers could be signalled using eight switches, by thinking of the switch positions as digits, \emptyset for off, 1 for on. This group of eight is called a byte, and this is why the number 256 is encountered so much in computing. Why a group of eight? It just happened: the early calculators were able to work with four bits at a time, and the step up to eight bits has lasted for quite a long while. Sixteen-bit machines are still not very common.

The way the bits in a byte are arranged so as to indicate a number is along the same lines as we use to indicate numbers normally. When you write a number such as 256, the 6 means six units, the 5 is written to the immediate left of the 6 and means the number of tens, and the two is written one more place to the left, and is the number of hundreds. These *places* indicate the importance or *significance* of a digit (see Fig. 19.3). The 6 in 256 is called the 'least significant digit', the 2 is the 'most significant digit'. Change the 6 to 7, and the change is one part in 256. Change the 2 to 3 and the change is one hundred parts in 256 – much more important.

Having looked at bits and bytes briefly, it's time to go back for a moment to the idea of memory as a set of switches. As it happens, we need two types of memory.

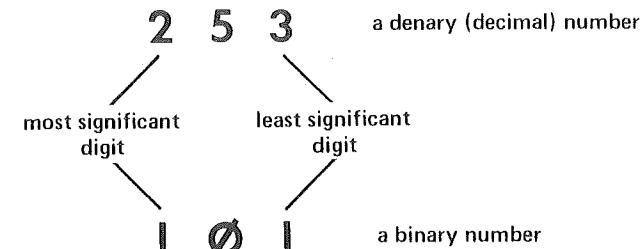


Fig. 19.3. Significance of digits. Our numbering system, unlike the old Roman system, uses the place of a digit to indicate its significance or importance.

One type must be permanent, like mechanical switches or fixed connections, because it is used to hold number-coded instructions that operate the computer. This is the type of memory which is called ROM, the letters meaning Read-Only Memory. The ROM is the most important part of your computer because it contains the instructions which make the computer carry out all of its actions. When you write a program for yourself, you store another set of number-coded instructions in a part of memory that you will want to be able to use over and over again. This is a different type of memory which can be 'written' or 'read', and if we were logical about it we would call it RWM, standing for read-write memory. Unfortunately, we're not very logical about it, and we call it RAM (Random Access Memory), which was a name used in the very early days of computing to distinguish this type of memory from one which operated in a different way. We're stuck with the name RAM now, so we'll have to make the best of it!

All done by numbers

Now let's get back to the bytes. We saw that a byte, which is a group of eight bits, can consist of any of 256 different arrangements of these bits, and that the most useful way of using these arrangements is to make each one represent a number in what is called binary code. The numbers are \emptyset to 255 (not 1 to 256, because we need a code for zero), and each byte of the 16,384 bytes of RAM in your Spectrum 16K can store a number in this range.

Numbers by themselves are not of much use, and we wouldn't find a computer particularly useful if it could deal only with numbers between \emptyset and 255, so we make use of these numbers as *codes*. Just as your Spectrum uses each key to do several different actions, each number code can be used to mean several different things. If you have worked with some BASIC programming, you will know that each letter of the alphabet, each of the digits \emptyset to 9, and each of the punctuation marks is coded as a number between 32 (which is the space) and 127 (which is the copyright sign on Spectrum). That leaves us with a large number of code numbers to use for other purposes such as graphics characters, and Spectrum, like all other small computers, uses most of the numbers also as codes for actions. When, for example, you press the key which is marked PRINT, what is placed in the RAM

memory of your Spectrum is not the sequence of ASCII number codes for PRINT, which would be 80,82,73,78,84, needing five bytes; but one single byte, 245. This single byte is called a 'token', and it can be used by the computer in two ways. One is to locate the actual characters which make up the word PRINT. These are stored in ASCII number-code form in the ROM, because they won't be changed (you don't want the word PERPLEXING to appear when you press the PRINT key!) and so that they don't take up space in your RAM. The other use of the token is to locate a set of instructions, also in coded form in the ROM, which will cause the action of printing (on the screen) to be carried out. These codes are the ones that we refer to as 'machine code', because they directly control what the machine does.

Practical interlude

As an aid to digestion, try a short program. This one (Fig. 19.4) is designed to reveal these keywords that are stored in the ROM, and it makes use of the BASIC instruction, PEEK. PEEK has to be followed by a number or a number variable,

```

10 PRINT 150;"      ";: FOR n = 150 TO 516
20 LET k = PEEK n
30 IF k < = 127 THEN PRINT CHR$ k;
40 IF k > = 128 THEN PRINT CHR$ (k - 128):
   PRINT n;"      ";
50 NEXT n

```

Fig. 19.4. A BASIC program to PEEK at words stored in ROM.

and it means: 'find the byte stored at this address number'. The groups of eight units of memory in your Spectrum are numbers from zero upwards, one number for each byte whether it is ROM or RAM, and because this is so much like the numbering of houses in a road, we refer to the numbers as *addresses*. The action of PEEK is to find out what number, which must be between 0 and 255, is stored at each address, and the Spectrum automatically converts the binary-coded numbers into ordinary (decimal, or more correctly *denary*) form. By using CHR\$, we can print the character whose code is the number we have PEEKed at. So far, so good. The program allocates 'n' as an address number, and then checks that PEEK n is less than 128 – in other words, that it is a character in ASCII code. If it is, it is printed.

Now the reason that we need the check is that the last character in each set of words or word is stored with a different coding. The number that is PEEKed for the last character is 128 + the ASCII code, rather than just the ASCII code. For example, the first three locations which the program PEEKs at, with addresses 150, 151, and 152, contain the numbers 82, 79 and 196. The number 82 is the ASCII code for R, 79 is N, and 196–128 = 68, which is the ASCII code for D, so this is where the word RND is stored. Why fiddle the D? The reason is that the Spectrum designers did not want to waste memory, so instead of having another

byte to separate RND from the next word, which is INKEY\$, they used this method of indicating to the computer where each word ends. When the Spectrum reads these codes one by one, it is programmed to stop reading when it comes to the one whose code number is greater than 128. We have made use of the same system in line 40 of the program of Fig. 19.4 to print the correct letter (by subtracting 128 before using CHR\$), and to print a space before moving on to the next letter.

Now for the next revelation. Take a look at the table of keywords that starts on page 186 in your Spectrum manual. Notice that they are in the same order as they are stored in the memory. By keeping them in order like this, with their token codes (starting with 165) also in order, it's easy for the computer to find one code if it is fed with the address of the start of the list – which is what it has to do each time you press a key.

Spectrum analysis

Now take a look at a diagram of the Spectrum, Fig. 19.5. It's quite a simple diagram because I've omitted all the detail, but it's enough to give us a clue about what's going on. This is the type of diagram that we call a 'block diagram', because each unit is drawn as a block, with no details about what may be inside. Block diagrams are like large-scale maps, which show us main routes between towns but

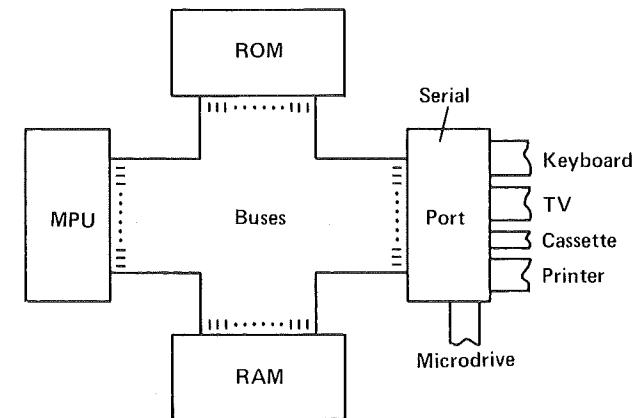


Fig. 19.5. A block diagram of Spectrum. The connection marked *Buses* consists of a large number of connecting links which join all the units of the system.

don't show side roads or town streets. A block diagram is enough to show us the main paths for signals in the computer, without the sort of confusing detail that you need if you want to show exactly what electrical connections are made.

Two of the blocks have already been introduced to you, ROM and RAM. ROM is the memory that can't be changed; it contains all of the essential instructions, along with keywords and token numbers, that are needed to make

the computer work. The RAM is used to contain your programs and a lot more besides, but we'll go into that later, in Part 6.

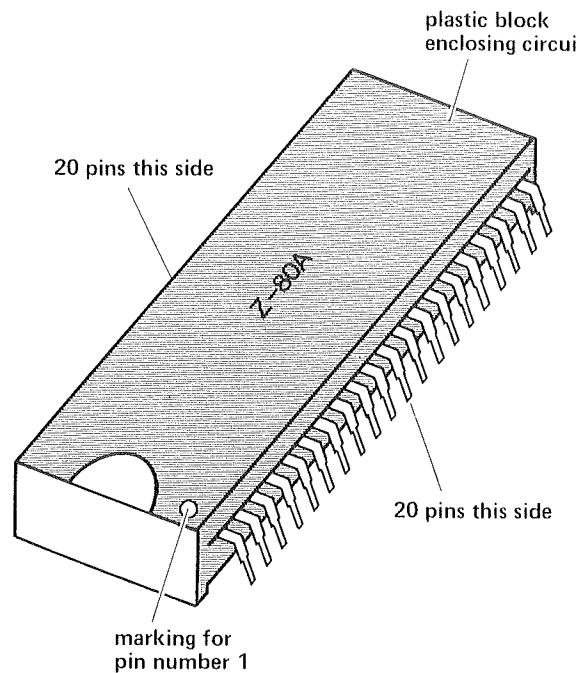


Fig. 19.6. The Z-80A MPU. The actual working portion is smaller than a fingernail, and the larger plastic case (52 mm long, 14 mm wide) makes it easier to work with.

The block marked MPU is a particularly important one. MPU means Microprocessor Unit (although some block diagrams use the letters CPU, meaning Central Processing Unit), and that's the main 'doing' unit in the system. Unit is a well chosen name in this case, because the MPU is just a single plug-in chunk, one of these silicon chips you read about, encased in a slab of black plastic, and provided with 40 connecting pins arranged in two rows of 20, as Fig. 19.6 indicates. There are several types of MPU made by different manufacturers, and the type which the Spectrum uses is called Z-80A. It is almost identical to the type called Z-80; the only difference is that the Z-80A can be operated more quickly if required.

What does the MPU do? The answer is practically everything, and yet the actions that the MPU can carry out are remarkably few and simple. The MPU can load a byte, meaning that a byte stored in the memory can be copied into another store inside the MPU. The MPU can also store a byte, meaning that a byte stored in the MPU can be copied into any address in the memory. These two actions (Fig. 19.7) are the ones that the MPU spends most of its working life in carrying out, and by combining them we can copy a byte from any one address to any other. You don't think that's very useful? That copying action, you see, is just what goes

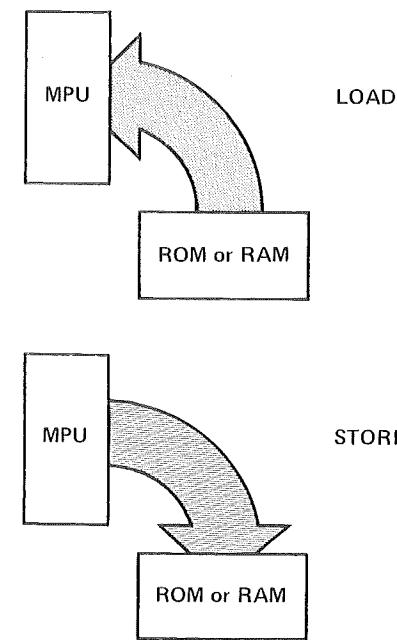


Fig. 19.7. Loading and storing. Loading means signalling to the MPU from memory, so that the digits of a byte are copied into the MPU. Storing is the opposite process.

on when you press the 'j' key and see the letter 'j' appear on the screen. The MPU treats the keyboard as one piece of memory and the screen as another and shifts bytes from one to the other as you type. That's a considerable simplification, but it will do for now.

Loading and storing are two very important actions of the MPU, but there are several others. One set of actions is the arithmetic set. Contrary to what you might expect, these consist of addition and subtraction only, and of no more than two-byte numbers either. How does the computer carry out arithmetic with larger numbers, numbers with fractions, how does it carry out multiplication, division, raising to powers, logarithms, sines and cosines? The answer is by machine code programming that is contained in the ROM. If these programs were not there, you would have to write your own, and a BASIC program for carrying out multiplication, using only addition, would be long and tedious, not a pretty sight.

There's also the logic set. MPU logic is, like all MPU actions, simple and obeys rigorous rules. Logic actions compare the bits of two bytes, and produce an 'answer' which depends on these bit values (0 or 1) and on the logic rule that is being used. The three logic rules are called AND, OR, and XOR, and Fig. 19.8 shows how they are applied.

Another set of actions is called the jump set. A jump means a change of address, rather like the action of a GOTO in BASIC, and it's the way in which the MPU carries out its decision steps. Just as you can program in BASIC:

100 IF a = 36 THEN GOTO 1050

so the MPU can be made to carry out an instruction at an entirely different address from the normal one, which would be the next address number. The MPU is a programmed device, meaning that it carries out each action as a result of being fed with an instruction byte which has been stored in the memory. Normally when the MPU is fed with an instruction from an address somewhere (usually in ROM), it carries out the instruction and then 'reads' the instruction byte that is stored in the next address up. A jump instruction would prevent this from happening, and would instead cause the MPU to read another address, one that was specified in the jump instruction. This jump action can be made to depend on some previous action, such as a zero, or positive, or negative answer to a subtraction, addition or comparison.

That isn't a great list, but the actions which I've omitted are not very important, nor very different from the ones in the list. What I want to emphasise is that the magical microprocessor isn't such a very smart device. What makes it so vital to the computer is that it can be made to carry out its actions very quickly, and each action is completely controlled by programming, sending it electrical signals. These signals are sent to eight pins, called the *data pins*, of the MPU and, as you will have realised, these eight pins correspond to the eight bits of a binary-coded byte. Each byte of memory will therefore be able to affect the microprocessor by sharing its electrical signals with the MPU. Descriptions in words like this take too long to write more than once, so we speak of reading and writing, always from the point of view of the MPU. Reading means that a byte of memory is connected to the MPU so that each 1 bit causes a 1 signal on a data pin, and each \emptyset bit causes a \emptyset signal on the corresponding data pin. Just as reading a paper or listening to a tape doesn't destroy what is written or recorded there, reading a memory doesn't change the memory in any way, and nothing is taken out. The opposite process of writing does, however, change memory. Like recording a tape, writing obliterates whatever existed there before, so that when the MPU writes a byte to an address in the memory, whatever was stored at that address previously is there no more and has been replaced by the new byte. This is why it is so easy to write new BASIC lines replacing old ones at the same line number.

Pick a number, any number ...

Do you really write programs in BASIC? It might sound like a silly question, but it's a serious one. The actual work of a program is done by coded instructions to the MPU and so far you don't write any of these. All you do is to select from a menu of choices that we call the BASIC keywords, and arrange them in the order that you hope will produce the correct results. Our choice is limited to the keywords that are designed into the computer in the ROM. We can't alter the ROM, and if we want to carry out an action that is not provided for in the ROM, we must either try to make it work by combining BASIC commands, or operate directly with machine code on the MPU. It's like the difference between talking of a 'motorised vehicle with a capacity for transporting more than eight persons', and a 'bus'. When you have to carry out actions with only a limited number of

AND

The result of ANDing two bits will be 1 if both bits are 1, \emptyset otherwise:

$$1 \text{ AND } 1 = 1 \quad \left\{ \begin{array}{l} 1 \text{ AND } \emptyset = \emptyset \\ \emptyset \text{ AND } 1 = \emptyset \end{array} \right\} \quad \emptyset \text{ AND } \emptyset = \emptyset$$

For two bytes, corresponding bits are ANDed

$$\begin{array}{r} 10110111 \\ \text{AND} \quad 00000111 \\ \hline 00000111 \end{array}$$

only
these bits
exist in both
bytes.

OR

The result of ORing two bits will be 1 if either or both bits is 1, \emptyset otherwise:

$$1 \text{ OR } 1 = 1 \quad \left\{ \begin{array}{l} 1 \text{ OR } \emptyset = 1 \\ \emptyset \text{ OR } 1 = 1 \end{array} \right\} \quad \emptyset \text{ OR } \emptyset = \emptyset$$

For two bytes, corresponding bits are ORed

$$\begin{array}{r} 10110111 \\ \text{OR} \quad 00000111 \\ \hline 10111111 \end{array}$$

only
bit which
is \emptyset in
both.

XOR (Exclusive-OR)

Like OR, but result is zero if the bits are identical

$$1 \text{ XOR } 1 = \emptyset \quad \left\{ \begin{array}{l} 1 \text{ XOR } \emptyset = 1 \\ \emptyset \text{ XOR } 1 = 1 \end{array} \right\} \quad \emptyset \text{ XOR } \emptyset = \emptyset$$

$$\begin{array}{r} 10110111 \\ \text{XOR} \quad 00000111 \\ \hline 10111000 \end{array}$$

if two bits
are identical
the result
is zero.

Fig. 19.8. The rules for the three logic actions, AND, OR and XOR.

commands, the result can be clumsy, especially if each command is a collection of other commands. Direct action is quick, but it can be difficult. The 'direct-action' that I'm talking about is machine code, and Part 6 will be devoted to understanding this 'language' which is difficult just because it's simple!

Take a situation to illustrate this paradox. Suppose you want a wall built. You could ask a builder. Just tell him that you want a wall built across the back garden, and then sit back and wait. This is like using BASIC with a command-word for 'build a wall'. There's a lot of work to be done, but you don't have to worry about the details.

Think of an option. Suppose you had a robot which could carry out instructions mindlessly but incredibly quickly. You couldn't tell it to 'build a wall', because these instructions are beyond its understanding. You have to tell it in detail, such as: 'Stretch a line from a point 85 feet from the kitchen edge of the house and against the fence, to one 87 feet from the lounge end of the house and touching the opposite fence. Mix three bags of sand and two of cement with four barrow-loads of pebbles. Mix in water until a pail filled with the mixture will just empty when held upside down. Fill the trench with the mixture ...'. The instructions are very detailed – they have to be for the brainless robot – but they will be carried out faultlessly and quickly. If you've forgotten anything, it won't be done, no matter how ludicrous it seems. Forget to specify how much mortar, what mixture and where to place it, and your bricks will be put up without mortar. Forget to specify the height of the wall as a number of layers of bricks and the robot will keep piling one layer on top of another, in the style of the Sorcerer's Apprentice, until someone sneezes and the huge wall falls down.

The parallel with programming is remarkably close. One keyword in BASIC is like the 'build a wall' instruction to the builder – it will cause a lot of work to be done, but not necessarily as fast as you would like. If you can be bothered with specifying the detail, machine code is a lot faster because you are giving your instructions to an incredibly fast but mindless machine, the microprocessor. We can stretch the similarity further. If you said to your imaginary builder – 'repair the car' – he might be either unwilling or unable, but a set of the correct detailed instructions to the robot would ensure that this task also was carried out. Machine code can be used to make your computer carry out actions that simply are not provided for in BASIC, though it's fair to say that many modern computers allow a much greater range of commands than early models, and this aspect of machine code is not as important now as it was then.

One last look at the block diagram is needed before we start on the inner workings of the Spectrum. The block which is marked Port covers a lot of circuits that are contained in one single chip, along with others. A *port*, in computing language, means something that is used to pass information, one byte at a time, into or out from the microprocessor system – the MPU, RAM, ROM parts. The reason for having a separate section to handle this is that inputs and outputs are important but *slow* actions. By using a port, we can let the microprocessor choose when it wants to read an input or write an output. For example, just imagine a BASIC program in which every line contained an INPUT. It would run very

slowly, because it will hang up and wait for you to press a key, followed by ENTER, on each line. If the program contained just one INPUT on the first line, it could then run uninterrupted by the keyboard until the end of the program. You will find, for example, that if you put the computer into an endless loop with:

10 GOTO 10

then no single key will have any effect on the computer. The computer still checks the port to see what's there each time it carries out the instruction, though, because if you interrupt the program by pressing CAPS SHIFT and SPACE together then you will break out of the loop. The use of the port, however, is a method of letting the computer get on with its work with only this type of interruption permitted.

In addition, there is no output from the computer except where we have commanded a PRINT or LPRINT, or when a program has come to an end. Once again we don't see anything new on the screen while the microprocessor is getting on with its work. The port isolates the section of the computer that deals with the screen, keeping whatever is there in place while the microprocessor deals with subsequent instructions. Without this isolation, your program would run much more slowly, and a lot of gibberish would appear on the screen each time an action took place. This is illustrated in the program shown in Fig. 19.9.

```
10 FOR n = 16384 TO 22528
20 POKE n, RND*255
30 NEXT n
```

Fig. 19.9. A program which POKEs on to the screen bytes generated at random.

Now the MPU isn't bothered where its information comes from, or what happens to that information once it has been sent out. It can even come from outside the computer, and go somewhere else outside the computer when the MPU has finished with it. You can see this for yourself every time you LOAD or SAVE a program on cassette! Instead of sending bytes to a different address in memory, your Spectrum simply sends them to the appropriate output ports. The difference has no significance for the MPU, but quite a lot for you! It means that your cassette recorder can now be fed with the individual codes that make up a program, byte by byte, and record them as audible signals on tape. There's more to it than that, but the basic idea is important. The keyboard is connected to the MPU by ports, and so is the TV, but there are plenty of spare port addresses, and, as you'll see, plenty of equipment you can connect through those addresses!

Input and output

Any device that uses the input and output facilities of your Spectrum, from a cassette deck to a household robot, operates on the same simple principles. As far as the computer is concerned, the information it is receiving from these devices and

sending out to them is just a series of switch positions coming from its normal addresses. At the moment you only have a limited number of things to attach to the output and input ports, but as you look through this section you'll see many other devices, including those mentioned in the introduction. Most of them plug into the user port at the back (see Fig. 19.10). Now you know something about the

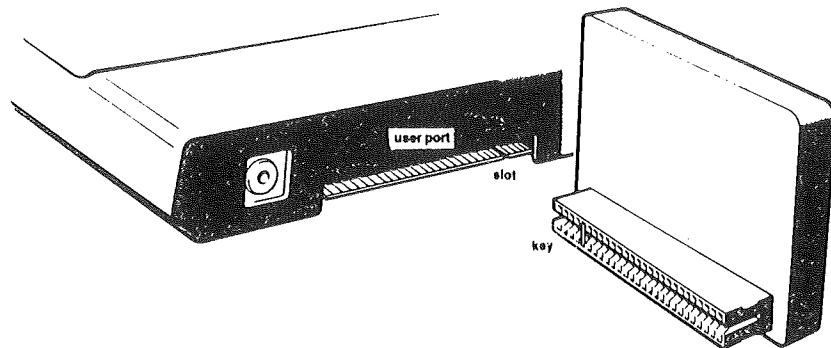


Fig. 19.10. Spectrum user port, and a typical add-on showing edge connector.

basic principles, they should look a little less mysterious! Digital tracers and lightpens simply convert drawings on paper or movements across a TV screen into numbers, which can be used by a program to create shapes and outlines on your TV screen. Sound synthesisers take number codes from your Spectrum and use them to make sounds: noise, music, or a close approximation to human speech. Robots simply convert code numbers into actual movements. Even this is only the beginning. Any device controlled by switches or by variations in electrical current can be controlled by your computer, and any device that sends out on/off codes or electrical currents can pass instructions and information into it. In fact the computer that runs your house in response to your spoken commands is no longer science fiction, it's available. If, for instance, you happen to be disabled, it's rather more than a clever toy.

Getting it together

Now that we've looked at the way computers and peripherals work together, the time has come to speak of *interfaces*. When I first became interested in computers I was convinced that the word 'interface' was a sales gimmick: an attempt to make me pay an outrageous price for a common or garden plug. However, an interface is usually a lot more than 'just a plug', and now perhaps you can see why it has to be. An interface has two very important jobs to do:

- To translate information sent by a peripheral into codes the computer can understand.
- To translate codes from the computer into information the peripheral can understand.

In other words, if the MPU is the telephone exchange then the interface is the translator who makes sure that the computer and the peripheral understand each other.

In practice this means different things for different types of equipment. The interface for your TV set looks, at first sight, very simple. All you can see is the socket for your aerial lead. What you can't see is the whole system of memory layouts and the special video chip that translate an ordinary set of computer codes into something that can be shown on a TV screen. That is all 'built into' your Spectrum, but it's as much a part of the TV interface as that socket. An interface is often more than a piece of hardware. It may include *software* – programmed instructions – too. This software can be part of the interface itself (rather like an extension to the Spectrum ROM, in which case it's known as *firmware*) or it may be a program you have to load into the computer before you can use a particular device. Either way, there's usually a lot more to using a peripheral than just slotting it in at the back! Some peripherals are almost like small self-contained computers. Speech synthesisers, for instance, usually have built-in circuits to translate codes from the computer into electrical impulses that can drive a speaker.

Getting down to brass contacts

That's enough theory for the moment! It's time to take a look at the different types of peripheral available for the Spectrum, and see just what they can do to help you. Each of the chapters that follows is pretty much self-contained, so if you're mainly interested in playing games, Chapter 20 should be right up your alley, and if you're a budding author who thinks computer games are for kids you can go straight to Chapter 26 (though I'd suggest you try a couple of games before jumping to conclusions!) Even if you don't think a particular chapter will help you much, take the time to skim through it, and do read the sections dealing with applications. A peripheral intended for games players may well have a serious business application, and another intended mainly for business users may help you create some superb arcade games. For the young among you (and the young at heart) let's start with the games.

Chapter Twenty

Add-ons for Games

It's perfectly possible to play arcade-type games on the Spectrum using the keyboard alone, but it isn't particularly easy. You may have bought Spectrum as a serious computer, but if certain users of any age from six to sixty get their hands on it, it will soon become a private arcade machine! If you feel that joysticks are 'just for games', it's worth remembering that joysticks are a simple and convenient way of manoeuvring *anything* around the screen, including cursors in word processing programs, graphics programs and large menus. You can even use them to help edit your own BASIC programs.

The basic Spectrum is not, of course, capable of handling joysticks directly: there is nowhere to plug them in and no interface to decode their signals. As a result, there's a bewildering number of different interfaces and joysticks available from a wide range of manufacturers. To spare you some of the bewilderment, I can tell you that any joystick with a D-type (Atari) plug, as shown in Fig. 20.1, will normally work with any Spectrum-compatible joystick interface that has the matching socket.

Joysticks – what they are and how they work

Most joysticks operate on one of two basic principles. The *switch* type just 'replaces' keyboard keys that would normally be used for controlling direction. You move your stick in any of the eight directions shown in Fig. 20.2 and it makes a connection inside the joystick. The interface translates that connection into the code for the equivalent key, or combination of keys, on the keyboard. If you like, it fools the CPU into believing that the key has been pressed. Some interfaces don't even go that far, but just provide a series of codes at a suitable input port that can then be used by a program in the computer. The **Kempston** interface, and others using this standard, adopt this approach.

The *analogue* joystick uses two *potentiometers* (an awful word for the voltage regulator that controls the volume on your radio). Up-down movement alters one potentiometer, and the other is altered by sideways movement. That doesn't mean you can only move in four directions! Movement in any direction can be broken down into vertical and horizontal parts – Fig. 20.3 should show you what I mean. Analogue joysticks are perfect for more serious applications, high resolution graphics, or games you want to program yourself, but very few commercially

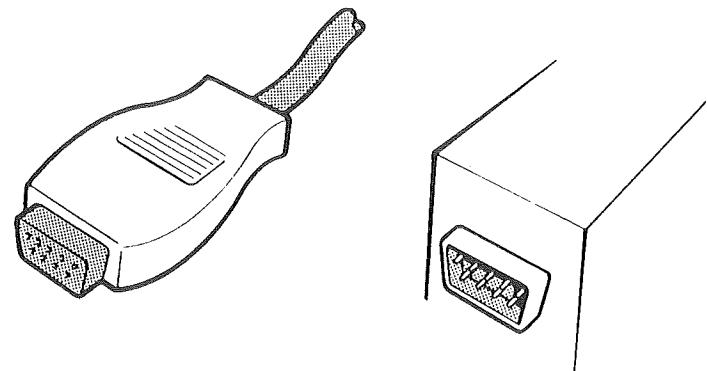


Fig. 20.1. Atari D-type plug and socket.

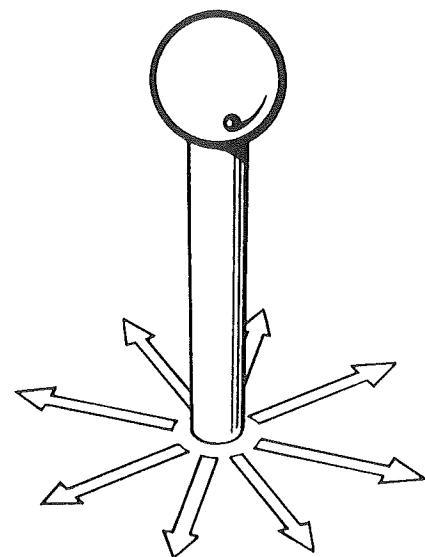


Fig. 20.2. A switch joystick can normally control movement in eight directions.

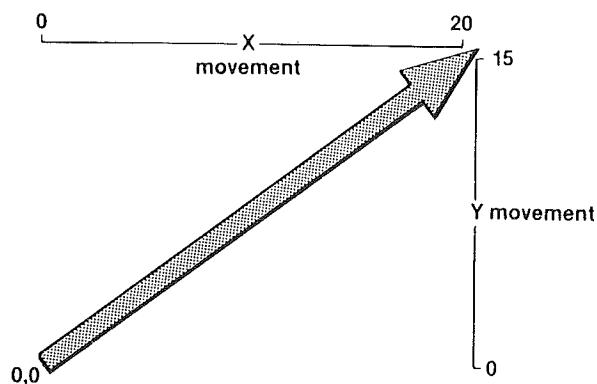


Fig. 20.3. An analogue joystick 'translates' movements into variations in horizontal (x) and vertical (y) positions.

available games are designed for them. However, control is far subtler than with the switch joystick, and certainly far more accurate.

There are a few joysticks that fall outside these categories; broadly speaking these are the gloriously simple and surprisingly effective mechanical joysticks from **Grant Design** and **EEC**, the revolutionary **Trickstick** from **East London Robotics**, **Le Stick**, which has no base (!), and the **Stack Light Rifle**, which could turn your living room into a shooting gallery.

Choosing your weapon

What are you looking for in a joystick? It isn't a silly question – different sticks vary enormously, and a lot depends on what you want to do with them. Check *your* choice for

- Comfort
- Responsiveness
- Accuracy
- Ruggedness

Comfort is important: any stick that gives you wrist-ache and callouses after half an hour is a bad buy. *Comfort* is an individual thing, though — how strong are *your* hands? *Responsiveness* is vital in any application, and some joysticks are decidedly sluggish. Surprisingly enough, a joystick that feels a bit stiff is often the best bet, but it shouldn't be too stiff for you to use over an extended period. It often helps if the stick is *self-centring*, which means that it returns to the centre position after you've moved it in any direction. *Accuracy* has a lot to do with responsiveness — if it's easy to feel when the switching has made a positive connection, it's easier to nudge the joystick into the very small movements required by games like *Splat* (to name but one). Again, self-centring sticks make nudging pretty easy. An inaccurate joystick, however comfortably styled, is not doing its job. *Ruggedness* is essential. Most joysticks will receive a great deal of rough handling, and if they fall to pieces after a few hard wrenches then you have good reason to be annoyed. Take a hard look at any joystick before you buy it, and make sure it will stand up to *your* use of it. Another word of warning — prices on joysticks vary widely depending on where and how you buy them. Sometimes you do better to buy one with an interface, and it can be worth your while to shop around. Prices given here should be taken as a rough guide. Incidentally, you will find that a number of different suppliers are releasing the same basic joystick, and some don't tell you that theirs is, for instance, the Atari or the Spectravideo. If in doubt, ask!

Out of the current offerings my personal favourite is the original **Quick Shot**, a self-centring switch joystick with a very comfortable trigger-style grip and four suckers that attach it firmly to any surface and allow you to use it one-handed. The firing button is on top, with a duplicate on the base to the left. The price is just £9.75. Joystick selection is a very personal thing, but others you might like to look out for are the new **Boss** from the USA, the more expensive **Kempston**

sticks, the **Superjoy**, and the **Sure Shot**. **Cambridge Computing** supply an excellent switch joystick with their **Intelligent Interface**, and the **Suncom JoySensor** is a touch joystick – yes, really! Suncom recommended it especially for handicapped users, but it does take a little getting used to.

Non-standard joysticks

From **EEC** and **Grant Design** come two similar and remarkably effective joysticks that have no electronic parts at all. They simply clamp over the keyboard and use a standard joystick stalk to operate the cursor keys. Although they can only be used on the basic Spectrum keyboard, and with programs that use the cursor keys, design is good, and they're surprisingly robust. The **Trickstick** from **East London Robotics** is a stubby cylinder fitted with six red knobs (see Fig. 20.4). The first

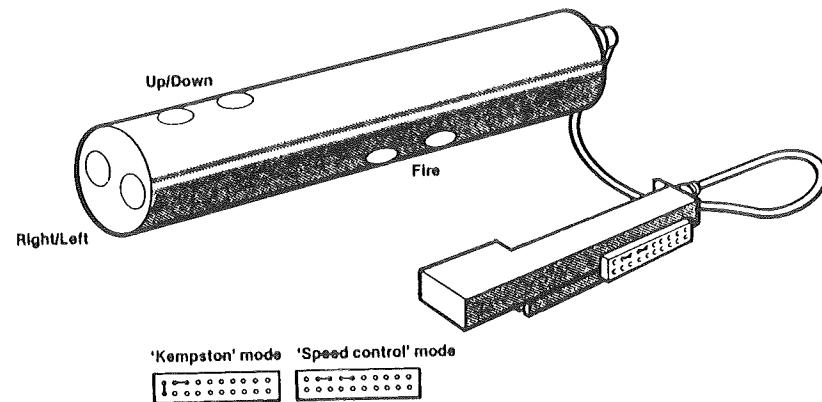


Fig. 20.4. The Trickstick and interface, with pin settings for Kempston mode and for its own software.

surprise is that it has no moving parts. The second is what it can do without them, as shown in the demonstration program supplied. With the Trickstick you can quite literally run rings around the opposition on screen. This ability to move graphics in smooth circles and curves under full control is a real innovation, and one that I hope to see more of. It's also possible to plug up to eight of them into one Spectrum. The stick, so I am told, is powered by a capacitative effect of the human body – apparently all God's children generate a 50 Hz hum.

The Trickstick has two major disadvantages. One is that the games to make full use of it have not yet been written. Second is its sensitivity, which is pretty difficult to adjust. This is important, because the Trickstick works by sensing how hard you are pressing its buttons, and if you get too excited you'll go further and faster than you ever intended to. A pair of curiously tiny and fragile looking miniature plugs on the interface allow the Trickstick to be converted into something like a standard joystick compatible with all games that are suitable for the Kempston interface (of which more in a moment). The same pins can be used to identify

which player is which in a multi-user game. In use, however, the stick seems difficult to control with accuracy, and it is badly let down by its software.

Le Stick is an equally curious-looking design, mainly because it manages without a base. To use it, you just tilt it in the direction you want to move! It works, so I'm told, by detecting the movement of mercury inside the stick itself; the mercury is heavy enough to make the necessary contact. My brief session with it convinced me that this was another stick that would take a little getting used to. The **Stack Light Rifle** isn't really a joystick; it's a reasonably accurate variant on the lightpen theme (see Chapter 25). It arrives looking like something Robert Mitchum used in *World War Two* – could come in useful for frightening burglars. The three programs supplied are extremely entertaining!

Introducing the interface

As mentioned earlier, the job of the joystick interface is to turn switch connections from the joystick into numbers, either the codes that would be sent by the corresponding keys, or an arbitrary set of codes recognised as a 'standard'. Joystick interfaces fall into three main groups according to the way they handle the problem:

- 'Fixed' interfaces. These allow the joystick to duplicate one (fixed) set of codes.
- 'Plug' interfaces. These use plugs or clips so that the user can choose which keys he wishes to duplicate.
- Programmable interfaces. These use on-board firmware or programmable chips called PROMs, and allow any joystick movement to duplicate almost any key entry in any mode.

Making the choice

Your choice, again, will depend on what you want to do with the joystick. 'Fixed' interface manufacturers like Kempston try to persuade the software houses to include the necessary code for their interface as an option in popular programs, solving the compatibility problem by diplomacy rather than flexibility. With the interface fitted you can also use joysticks in your own programs (see Fig. 20.5). If a commercial program doesn't include an option for your interface it may often include another where you pick the keys you want to use, so just pick the keys the interface duplicates. Normally you can do this just by moving the joystick up, down, left and right (or whatever) in response to the screen prompts from the program. Some interface manufacturers even supply 'conversion tapes' that will set up the computer to play a game that normally uses a different set of keys.

```

10 LET d=0: LET x=127: LET y=8
8
20 INPUT "Paper colour? ";p: IF
p<0 OR p>7 THEN GO TO 20
30 INPUT "Ink colour? ";i: IF

```

```

i<0 OR i>7 THEN GO TO 30
40 INPUT "Border colour? ";b:
IF b<0 OR b>7 THEN GO TO 40
50 PAPER p: INK i: BORDER b: C
LS
60 IF IN 31=2 AND x>0 THEN L
ET x=x-1
70 IF IN 31=1 AND x<255 THEN
LET x=x+1
80 IF IN 31=4 AND y>0 THEN L
ET y=y-1
90 IF IN 31=8 AND y<175 THEN
LET y=y+1
100 IF IN 31=10 AND y<175 AND
x>0 THEN LET y=y+1: LET x=x-1
110 IF IN 31=9 AND y<175 AND x
<255 THEN LET y=y+1: LET x=x+1
120 IF IN 31=6 AND y>0 AND x>0
THEN LET y=y-1: LET x=x-1
130 IF IN 31=5 AND y>0 AND x<2
55 THEN LET y=y-1: LET x=x+1
140 IF IN 31=16 THEN LET d=d+1
200 IF d=1 THEN PLOT x,y: GO T
0 60
210 IF d=2 THEN LET d=0: GO TO
60
220 GO TO 60

```

Fig. 20.5. Simple sketching program for the Kempston joystick. Move the stick and press the fire button once to draw, again to stop.

However, this is not going to help you with software that was not originally designed for use with joysticks; for that, you'll need something a little more sophisticated.

One particular 'fixed' interface needs a little extra description – **Sinclair's Interface 2**. Interface 2 has picked up a lot of flak in the computer press, largely because it could have been part of Interface 1 (see Chapter 23). What makes it special is the socket at the top for plug-in ROM cartridges. If this sentence baffles you take a look at Fig. 20.6, which shows the Interface 2 with a cartridge plugged in. The ROM cartridge replaces the computer's own ROM, turning even a 16K Spectrum into a perfect games machine that will actually handle games originally written for the 48K. With this system you will also be able to 'plug in' computer languages other than BASIC when (and if) the necessary cartridges are released. You can also remove the cartridge and use the interface as a conventional fixed interface in your own programs.

There are snags, though. Prolonged use may lead to the same kind of wobble I used to get on my ZX81 RAMpack, and if this crashes an arcade game just as you

are about to reach the score of the century you won't be pleased. Secondly, you have to turn off the power every time you change a cartridge. Thirdly, the user port at the back will *only* support a ZX printer. Finally, £9.95 for a ROM cartridge is a bit on the pricey side, and some are still £14.95.

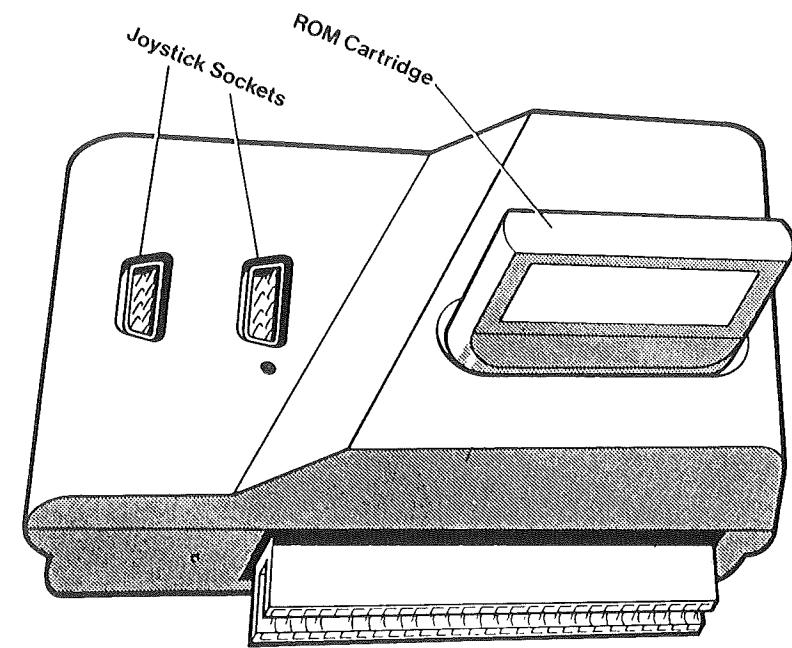


Fig. 20.6. The Sinclair Interface 2 with a ROM cartridge plugged in.

Plug interfaces are versatile and practical. They're also simplicity itself to program. Just take a colour-coded or labelled set of leads (usually UP, DOWN, LEFT, RIGHT, and FIRE), and fit them to sockets or leads corresponding to the keyboard keys you want to duplicate. However, these interfaces aren't usually very pretty (if that matters to you) and there are certain keys they can't duplicate (such as SYMBOL SHIFT D, for instance, which might be useful in some word processing programs). You will also have to swap all the leads around every time you use a program with a different set of command keys.

Programmable interfaces are extremely versatile and usually very easy to set up, though they are, of course, more expensive. What you're paying for is flexibility (and some fairly sophisticated electronics!) so you must decide for yourself whether you really need the facilities this type of interface can give you.

Buying guidelines

Among the fixed interfaces, there is little to choose between the **Kempston**, the **Protek**, and the older **Datel** units: the **AGF II** is cheaper and unlike the others has a ZX81-type through bus. The new **RAM Electronics Turbo** is setting out to offer

Interface 2 some competition with a ROM cartridge socket, full Spectrum through bus, and a neat arrangement to avoid the need for turning off the computer while changing ROM cartridges. Among the plug interfaces the **AGF I**, the **Jiles**, and the **Pickard** controller are worth a look. Among the programmables my personal favourite is the **Jiles**, closely followed by the **Cambridge Intelligent Interface** and the **Fox**, which carries its programming on board thanks to a handy rechargeable battery built into the unit. If these are too expensive for you then the **Stonechip** might be of interest, but it can be a laborious business to program it. A new programmable interface from **AGF** uses program cards - look out for it.

Points to watch

Most joystick interfaces only allow you to fit one stick. A few let you fit two. Again, as noted above, several joysticks provide an extra 'rapid fire' button, and not all interfaces can respond to this button. It's also worth checking if the interface is fitted with a through bus that carries all its connections through to a socket at the back so you can add other peripherals behind it. If it is, see if it uses the full width of the Spectrum user port; if it doesn't, it won't carry *all* the connections through. Narrower connectors of this type also fit the ZX 81 user port, which is smaller, so peripherals designed for the wider Spectrum port can't be fitted behind them. This is something that happens quite often with Spectrum peripherals, and it's less serious than it seems for that very reason. There are dozens of other peripherals that also use the ZX81 connector!

Talking of other peripherals, do be careful about choosing and using joystick interfaces with replacement keyboards. Some, such as the **Stonechip**, the **Kempston** and the **Cambridge Intelligent Interface** won't fit comfortably behind some of the new replacement keyboards described in the next chapter, so think about this when you're making your choice. However, all of them will normally fit if Interface 1 is connected outside the keyboard.

One final point. Quite a number of other peripherals include a joystick interface as a sort of extra bonus. These interfaces are always the 'fixed' type, and rarely allow you to fit more than one joystick. Usually they use spare space on another chip, such as a speech or music chip. For the occasional games player who is more interested in music, a unit of this type could save money. You'll find full details in Chapter 24.

Using your joystick

If you're going to use a joystick in your own programs then it's important to understand a little about how it works, and for that you'll need to understand how the *keyboard* works, which should make a useful introduction to the next chapter! The Spectrum keyboard, as you can see in Fig. 21.1, divides each horizontal row of keys into two half-rows. Each of these can be 'read' using an IN command, as follows:

| | |
|-----------------|----------------|
| CAPS SHIFT to V | PRINT IN 65278 |
| A to G | PRINT IN 65022 |
| Q to T | PRINT IN 64510 |
| I to S | PRINT IN 63486 |
| 6 to 0 | PRINT IN 61438 |
| Y to P | PRINT IN 57342 |
| H to ENTER | PRINT IN 49150 |
| B to SPACE | PRINT IN 32766 |

That probably looks fairly baffling, so connect up your joystick interface and try the following program

```
10 REM Joystick
20 PRINT AT 12,0; IN 61438;" Half row 6 to 0"
30 PRINT AT 14,0; IN 63486;" Half row 1 to 5"
40 GO TO 20
```

Now try moving your joystick and watch the numbers change. Press the keyboard keys in the appropriate half rows and you should be able to match these numbers. Notice also that you can actually press more than one key, and get a usable number. For instance, if you press keys 7 and 8 together you should get the number 243. These two keys are also the cursor keys for UP and RIGHT, so you can use the reading 243 in your programs to send something diagonally upwards to the right. For more details about the actual codes, look at Chapter 35. Elsewhere in this book you'll find more joystick routines, but for the moment I'd like to concentrate on what the joystick and its interface are simulating - that much abused and totally necessary workhorse, the keyboard.

Chapter Twenty-one

New Keys for Old

If you are happy with the keyboard on your Spectrum, fine: skip this chapter and go on to something that interests you. Before you do so, however, think about how the Spectrum keyboard is made. The earlier ZX80 and ZX81 computers both used 'touch sensitive' keyboards that were not, in fact, all that sensitive. The only way to be sure you had actually entered a character was to look at the screen and see if it was there! The Spectrum keyboard, believe it or not, is only a little more sophisticated: underneath that cosmetic exterior there beats the heart of a ZX81! The keys are merely lugs on a single rubber sheet sandwiched between the metal cover and the keyboard circuitry underneath (see Fig. 21.1.). When you press

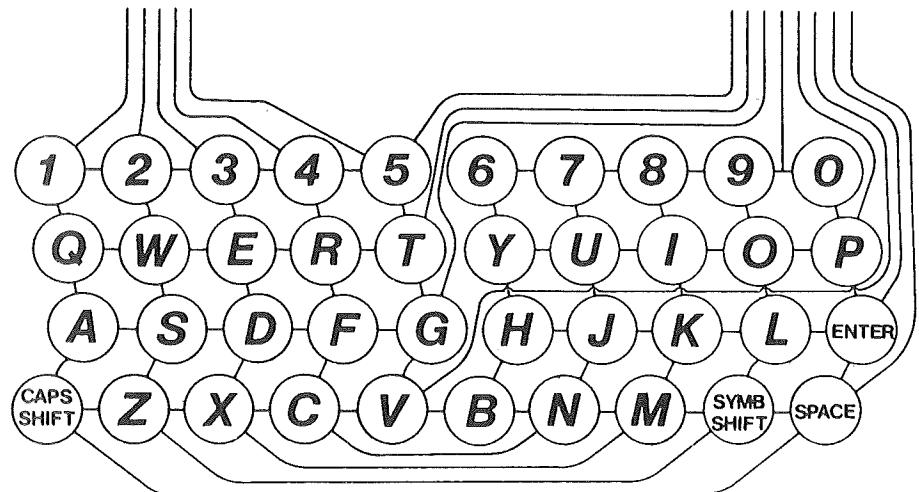


Fig. 21.1. The *real* Spectrum keyboard! Notice that five wires go to the ribbon cable on the left, and eight to the one on the right.

down on one of these lugs, two wires are pushed into contact with each other. Depending on which connections have been made in this way, a code is generated and sent to the input port for the keyboard. This is hardly the way to make a professional quality keyboard, as Sinclair themselves would be the first to admit – in fact the keyboard is one of the items that helps keep down the price of the Spectrum.

Even so, the Spectrum keyboard can be used for word processing. It's slow, and not particularly comfortable, but the first part of this section was written in this

way. However, in due course you may, like me, come across a more serious problem. Round about the end of Chapter 19 my beloved Spectrum suddenly gave up with typing the letters m, n, and b. It wasn't too keen on SYMBOL SHIFT, either. Notice that all these keys are on one half-row. When the first of the replacement keyboards I was testing arrived, I connected it up and found that the computer's main circuitry was perfectly OK. The trouble was purely and simply the keyboard.

If you are quite sure this could never happen to you, then don't be. For one thing the half-row from B to SYMBOL SHIFT on my Issue 2 is right on top of the heat sink, and obviously long and hard use of the computer will have an effect on the keyboard connections immediately above. Within a week of my keyboard going down I had heard of three other Spectrum keyboards going down in exactly the same way and on the same row. The villain of the piece does indeed seem to be the heat sink – its near presence causes slight deformation of the keyboard, and the contacts become dirty until they eventually fail to operate. It's a fault that can be put right, but at a price, and by leaving you without a computer for a while.

The Spectrum keyboard is cheap, and it's perfectly adequate for an inexperienced beginner, but it will wear out, and if you're becoming a more serious user you may very well want something better anyway. So what should you be looking for?

Looking for the perfect keyboard

Everyone's idea of the perfect keyboard is going to be a little different – it depends what you want to use it for. The main uses for a replacement keyboard are

- Word processing
- Entering data (figures or words)
- Entering programs
- Playing games

For *word processing* – writing books like this one, for instance – the keyboard should be as much like a typewriter as possible. You are looking for:

- 'Dished' key tops that will catch and hold your finger to stop it slipping.
- A 'raked' keyboard, which presents the keys to you in stepped ranks rather than all parallel to each other.
- Sprung keys that move when you touch them (so that you're sure they have responded) and are not too tiring to use.
- A proper space bar (vital).
- An extra CAPS SHIFT key on the right-hand side (vital).
- Extra 'single-entry' keys that will let you enter punctuation (commas, full stops, colons, semicolons, etc.) without pressing another key such as SYMBOL SHIFT first.
- A single-entry DELETE key. Most people use this key more than any other! Full compatibility with Microdrive – you're going to need this for any serious word processing system unless you opt for disk drive instead.

For *entering data* you really need all the features mentioned above, plus a couple more:

- A numeric keypad, like the keypad on a calculator, is invaluable for entering long strings of numbers.
- A separate full stop key is vital for adding decimal points, and should ideally be on the numeric keypad.

If your main use of the keyboard is *entering programs*, then you can probably do without some of the extra punctuation keys and you may even manage without the space bar. However, useful extra features might include:

- Single-key entry to extended mode (so you don't have to press CAPS SHIFT and SYMBOL SHIFT together just to reach the red and green keywords).
- Very clear key markings (so you can find rarely used commands as easily as possible).
- Single-entry cursor keys (so you don't have to press CAPS SHIFT first).
- Easy connection with Interface 1 and with the full range of add-on units for the Spectrum.

Figure 21.1 shows some of these extra keys on the Transform keyboard.

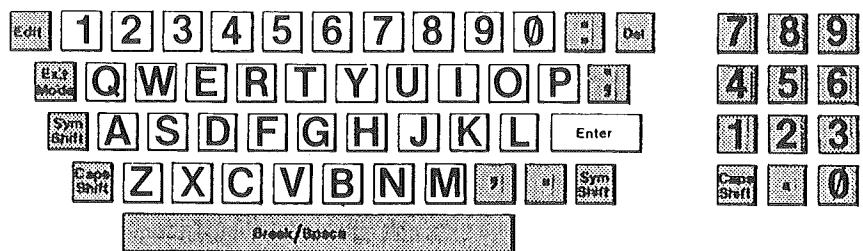


Fig. 21.2. Layout of the Transform keyboard. Shaded squares indicate extra keys.

If you're *playing games* you're usually much better off with a joystick! However, if you are an occasional games player with no wish to buy a joystick then look out for

- A numeric keypad (duplicating the cursor keys), or
- Single-entry cursor keys.

At the moment these are the **Fuller FDS**, the **dk'tronics**, the **AMS**, the **Transform**, and the **Saga Emperor** (though on the Emperor, AMS and the dk'tronics you'll have to hold down CAPS SHIFT as well).

The choice is yours

None of the keyboards on the market at the time this book was published had all the features listed above, though the promised top model in the Saga range may well do. Until then you'll have to choose according to the features that are most

important for your particular use of the Spectrum. Generally speaking you have two options:

- An expensive keyboard (£50 plus) which gives advanced features such as single-entry keys for DELETE, EXTended mode, cursor, etc., and may include a numeric keypad.
- A cheaper keyboard that effectively duplicates the layout and facilities of the Spectrum keyboard without offering anything extra (apart, of course, from more comfortable keys and in some cases a space bar).

The cheaper keyboards can cost as little as £30, but their lack of facilities can make this poor value.

Among the more expensive keyboards the leader for my money is the new **Saga** range – the **Emperor** model has an unmatchable array of extra keys for word processing, data entry, and programming. The **Transform** is good but expensive. Worth looking at are the **dk'tronics** board, recently released with a space bar and some single-entry keys, and the **Fuller FDS**, which has several superb extra keys including two to give immediate access to the two different EXTended modes.

Among the cheaper boards your best bet is probably the **Fuller FD42**. The **Ricoll** has a metal case, a space bar, and a spare CAPS SHIFT key but it's difficult to fit Interface 1.

Keyboards and Interface 1

Fitting Interface 1 can be another fertile source of problems with replacement keyboards, and one I am glad to say the more go-ahead manufacturers are already dealing with. The **Transform** and **dk'tronics** boards in particular are excellently designed for Interface 1 – the **Fuller**, at the time of writing, is very poorly designed for it, and the last set of instructions I saw invited customers to cut the necessary apertures and holes for themselves (this on a keyboard costing nearly £50!) Many keyboards are designed to allow fitting of Interface 1 outside the keyboard case, and this is not always a bad thing – in the case of slimline keyboards like the **Saga** and the **Lo-Profile** it gives useful added tilt. However, if you plan to use Interface 1 with your chosen keyboard do make sure there are no connection problems before you buy!

Many replacement keyboards can only be fitted by taking your Spectrum, and sometimes even Interface 1, out of their cases. It's easy enough, but it will invalidate your guarantee, so don't do it lightly. *Always start by unplugging the Spectrum at the mains.*

When you have undone all the screws (you'll need a screwdriver with a 'Philips' or Pozidriv type head for this) the keyboard simply lifts off to reveal the main PCB (printed circuit board) – see Fig. 21.3. Coming from the back of the keyboard you'll see two ribbon cables that plug into connectors on the PCB. All you have to do is tug (gently) at each ribbon cable, and it will slip out of the connector. To

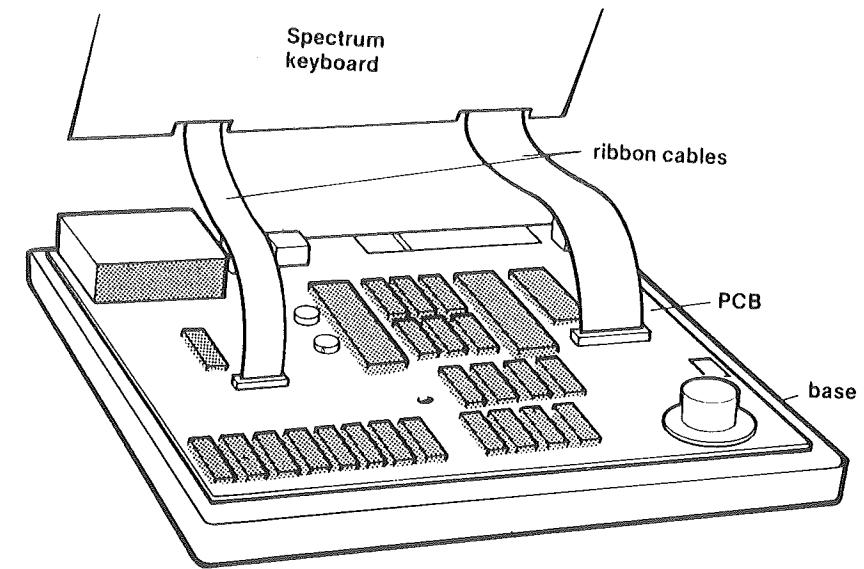


Fig. 21.3. Fitting the new ribbon cables to the Spectrum PCB.

remove the PCB from the lower casing, just undo the small screw near the centre: you should be able to lift it out easily. Taking Interface 1 out of its case is just as easy, but be particularly careful when you are easing the edge connector out of the hole in the top of the case: there's a row of fragile looking metal wires just behind it, and it is very easy to put damaging pressure on them. Breaking any of them will ruin the Interface, and invalidate the guarantee!

In the case of the **Fuller** you need to plug your power supply directly into the keyboard. A feeder wire takes it down to the Spectrum's normal power socket. I am told this can cause trouble when running some machine code programs, but I can't say I've noticed this myself. If you haven't taken your power unit out of its case and refitted it inside the **Fuller** case (and frankly I wouldn't recommend it) you may hit a snag at this point. You will have to find somewhere to pass the two power cables, the one from your power unit and the one from the keyboard, through the casing. No provision is actually made for this if the PCB is mounted inside the case, so again it may well be a question of drilling your own holes. It may be better to pay the extra for a linking cable (about £10) and leave the Spectrum in its case.

In the end, choosing a keyboard is a matter of personal taste, and depends very largely on what you want to use it for. I hope that these brief reviews will at least give you some idea of the ones you are most interested in. Remember you are going to live with your choice for quite some time, so don't be bullied or cajoled into buying a keyboard that you are not quite sure about.

Chapter Twenty-two

Enlarging the Memory

This chapter is purely for 16K Spectrum owners who want to get out into the wider world of 48K computing. It is possible to add extra memory to the 48K, but frankly you'll be better off with a storage system like Microdrive. However, if you want to upgrade your 16K to 48K you have four ways of doing it:

- Ask Sinclair to do it for you.
- Ask a competent dealer to do it for you.
- Buy an upgrade kit and do it yourself.
- Buy a plug-in memory unit.

Each of these options except the first and last involves a series of steps that could well make your blood run cold. The 'upgrade' is in fact rather less simple than it sounds! The first step is to open the case. As we saw in the previous chapter, this immediately invalidates the Sinclair guarantee. Next you have to take individual microchips and fit them to the main printed circuit board (PCB for short, if you skipped Chapter 21). If you've never actually seen a microchip before, Fig. 22.1 will show you what you're up against. Most of the chips in the upgrade kit are about $\frac{3}{4}$ inch long.

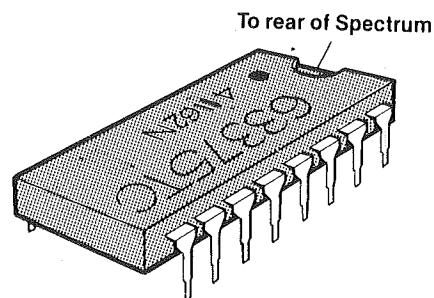


Fig. 22.1. A microchip.

Careless handling at this stage could damage one of the chips (annoying and time wasting) or even, if you were very unlucky, the computer's own PCB (disastrous!). Assuming that all is well, however, it's still possible to insert a chip the wrong way up. At best, this means your upgrade won't work. At worst, you could permanently damage other chips on the main PCB. A good upgrade kit, of

course, comes with clear instructions and detailed diagrams, but unless you are very confident that you can cope, beware of carrying out upgrades yourself. For those who want to, there are detailed instructions at the end of the chapter. All the same, this is undoubtedly the cheapest way to reach the magical 48K. Many Spectrum upgrade kits are sold to dealers rather than direct to the general public; in any case, the only thing that's really different about upgrade kits is the quality of the instructions. Most kits sell for around £25. Much the same situation applies to plug-in RAM units. Prices are pretty stable for these units at about £40. They either work or they don't: if they don't, send them back and demand one that does, or your money back. Even the best hardware companies sometimes have a bad day.

Going to Sinclair

The main advantage here is peace of mind – your Sinclair guarantee remains unaffected. Snags are the price – currently around £40 – and the delay, which will be at least two to three weeks. For £40 you could actually buy a plug-in memory unit and have your 48K Spectrum instantly, so unless you're desperate to have your memory unit inside rather than outside the case, this really isn't the best option.

Going to a dealer

Snag number one is finding a *competent* dealer you know you can trust. For this it's hard to beat personal contact and personal recommendation, so talk to your local User Group. If you don't know where they are, look in the Sinclair-dedicated magazines like *ZX Computing* or *Sinclair User*, or check at your local library. If fitting the upgrade is going to mean invalidating your Sinclair guarantee, check that the dealer who does the job is offering some guarantees of his own.

Doing it yourself

To check how easy this was I looked at two kits, one from Fox Electronics and one from dk'tronics. The Fox kit arrives in a clear plastic box, with the chips bedded on polystyrene foam and the instructions folded above them. It's very well documented, and good value. The dk'tronics kit arrives with the chips already in place on a neat diagram of the main PCB – ten out of ten for presentation. Both are intended for Issue 2 machines. At the time of writing, Issue 3 16K machines are not all that common in the UK, but in any case you will find the issue number printed in white ink along the bottom of the PCB. If you have an Issue 3 16K then the upgrade kit from East London Robotics should fit it.

Getting ready

The first thing to watch out for is static electricity. It has a nasty habit of

scrambling the information that's been programmed into your microchips. To avoid such disasters:

- Wear nothing on your forearms.
- Sit still at the table for a while before opening the upgrade pack.
- Don't shuffle your feet on the carpet while you're working.

Now open up the Spectrum case as described in Chapter 21. If you like, you can remove the keyboard altogether by *gently* unplugging the ribbon cables from the sockets on the main PCB. The dk'tronics kit contained dire warnings against this, but I can't see why. You have to do this to fit their keyboard! If you know look at the main PCB you should be able to see twelve empty spaces just begging to have microchips plugged into them. Fortunately eight of them, the memory chips, are exactly the same! For that reason, if nothing else, start with the memory chips.

Fitting the chips

Priority one is to get the chip the right way round. Figure 22.1 should help here – the half-moon shaped cutout or darker dot should be at the back, pointing to the Spectrum's rear connections. Some chips use both markings. The instructions in your kit will tell you which are the memory chips – they're identified by the number printed on the chip, though confusingly enough some manufacturers add extra numbers before and after the identifying code. Take your time about this.

Once you know which chip you want, check that you know where to put it. The memory chips go in the eight empty spaces near the front of the PCB (see Fig. 22.2). Pick up the first one, but be careful not to grip it so that you're squeezing the

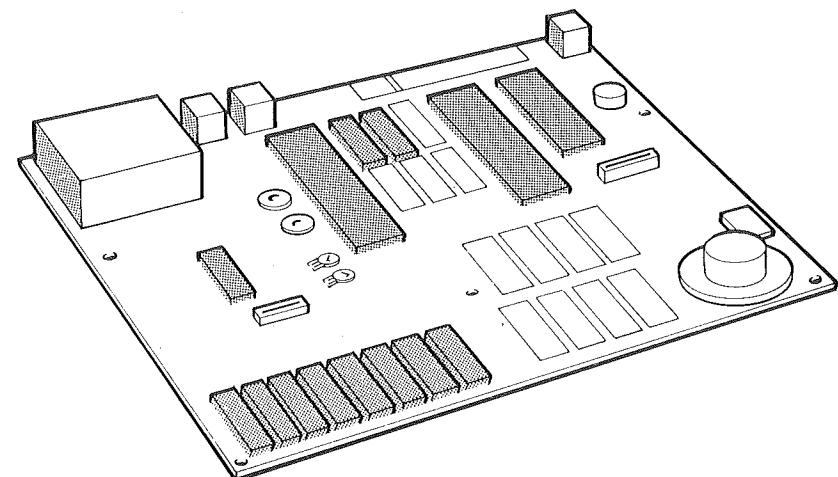


Fig. 22.2. 16K Spectrum PCB showing spaces for new chips.

pins – they bend! Hold it with your thumb pressing the front end and your finger pressing the back end, and keep it absolutely level with the PCB as you position it

over the socket. If the pins are splayed out, and won't fit in the socket, don't try to bend them with your fingers. Lay the chip on its side, with the pins resting flat on the table, keep your grip on the front and back, and push the chip *gently* until the pins are at right angles to the body of the chip itself.

When fitting the chip, use the forefinger of your other hand to do the pushing while you steady the chip with the grip described above. Push it in *evenly* – bending the pins could cause serious damage – and check that all the pins are properly located. If one of them isn't, you could easily bend it completely out of shape by pressing down on the chip. If you've made a mistake, or something goes wrong, lever the chip out gently with a screwdriver, a very small amount at a time, starting at one end and switching to the other so that the chip remains as level as possible all the time.

The remaining four chips are all different. Fit them according to your supplier's instructions, and make sure they are all in the right place and the right way round. Check all the chips this way, and check them again. At this stage it's difficult to be too careful. Some upgrade kits require you to cut wire connections on the PCB. If yours does, take great care with it! When you're satisfied that all is as it should be, replace the keyboard and close the case.

Testing your upgrade

Now comes the moment of truth. Make sure the TV is connected and switched on, then plug in the power lead at the back of the computer and switch on at the mains. If you don't get the normal black screen and copyright notice within four seconds then switch off – *fast!* Something is obviously very wrong. Open the computer and double-check your connections. If you can't find the error, then there are two nasty possibilities:

- The upgrade kit is faulty in some way.
- Your computer's PCB is faulty or damaged.

In either case you'll have to remove the upgrade chips – carefully! – and pack them up again. Send them back to the supplier for checking. Meanwhile, once your computer is back to its old 16K self, plug it in and check it out again. If the fault persists, then you'll have to get *it* repaired as well – and hard luck. If all is well on the first test, and you get a copyright notice, then try entering:

```
10 LET memory=(PEEK 23732+256*
PEEK 23733)+1
20 LET size=memory/1024-16
30 PRINT size
```

Fig. 22.3. Simple memory check routine.

When you RUN this program you should see the magic figure 48 appear on screen – not an acid test that all is well, but a very good indication that it might be.

Your upgrade kit should normally include a fuller test program that tests out the available memory completely. If it doesn't, try this program:

```
10 CLEAR 32000: LET test=0
20 FOR m=32768 TO 65535: POKE
m,test: PRINT AT 0,0;m: NEXT m
30 FOR m=32768 TO 65535: PRINT
AT 0,0;m: IF PEEK m>>test THEN
60 TO 70
40 POKE m,255: NEXT m: IF test
THEN 60 TO 60
50 LET test=255: GO TO 30
60 PRINT "Relax. Everything ch
ecks out": STOP
70 PRINT "Sorry, There's a fau
lt at ";m
```

Fig. 22.4. Full memory check routine.

Don't sit by the screen. It will take quite a while so go and make yourself some coffee while you wait. All being well you should get the friendly message in line 60. If not, you'll have to remove the upgrade it and send it back for checking.

Plug-in RAMpacks

This is the easy way out, though not the cheap way. Prices are typically about £40 for a plug-in pack, compared with £20–£25 for an upgrade kit. Again, you're paying for convenience and peace of mind. However, check that your plug-in has a proper through bus (all the ones I've seen had one) and check that it's a snug, secure fit in the user port or behind Interface 1. That's right, *behind*. That may sound odd, but when I tried putting one between the Interface and the computer, where I'd expect it to be, I achieved a fairly spectacular lock-up. A tight fit is important; loose fitting add-ons will crash your computer with monotonous regularity.

Some manufacturers offer double-size RAMpacks that give your computer up to 64K of RAM. Before you go wild with excitement, I should point out that you can't get to all 64K at once – you have to use a 'switching command' which effectively shuts out one 'page' of the extra RAM and cuts in another. This is because the 48K Spectrum can only cope with 65536 addresses; it has no way of reaching addresses with a higher number. Units like these may sound attractive, but they are really for machine code programmers; it's hard to use them properly from a BASIC program. The remarkable range of add-on units from Basicare, which could theoretically give you *several thousand* K of 'paged' memory (see Chapter

29), is also for serious users – and it's rather complicated. You'll really be better off with a simple fast access storage system like Microdrive. Talking of which, it's about time to move on to a closer look at storage systems in general, and see what's available and what is likely to suit your special needs.

Chapter Twenty-three

Cassette Decks, Microdrive and Disk Drive

If you are just starting out with your Spectrum, then the idea of hundreds of kilobytes of storage space may seem irrelevant to you. After all, what would you use it for? The answer will soon become very clear – you would use it for the enormous mass of part programs, junked programs, and the odd one or two successful programs that everyone seems to generate in their first few weeks at a Spectrum keyboard. The moment you put together your first program and save it on a cassette tape your problems are beginning. In fact one of your early priorities should really be a database program. You may not think that you are going to need a special program to find your programs, but believe me you will.

The trouble with cassette storage is speed (or rather lack of it) and accessibility. On a long tape, the only way to get to the program you want is by wading through a good many others that you don't want. Of course, you can buy much shorter cassette tapes – and finish up swimming in them with nowhere to put the furniture or, come to that, books like this one. So what are the options?

The filing file

For Spectrum owners there are three main choices:

- Cassette tape
- Microdrive
- Disk drive

Cassette tape is slow to LOAD and SAVE, and difficult to access quickly. It can also be unreliable. Normally you *must* VERIFY to make quite sure you've got a decent copy of your program. However, this is ultra-cheap storage; it uses hardware that most people already have, and cassettes are available on any high street.

Microdrive provides rapid access to about 90K of information stored on a tiny tape cartridge. A special drive unit is needed, and Interface 1 must be fitted to your Spectrum before the unit can be used. Up to eight drive units can be chained together. Typical loading time for a BASIC program is seven seconds. The cost for a starter system including Interface 1, one drive, and two blank cartridges is about £90 if you buy direct from Sinclair.

Disk drive, the serious user's storage system, is now available for the Spectrum. It allows extremely fast access to at least 2100K. However, it's expensive compared

to the cost of the computer itself. Prices start at about £100, including the necessary interface and in some cases a suite of software.

One of these systems will be right for your particular needs – and for your particular pocket. But if you feel that even Microdrive is way above your budget, don't despair. Cassette software isn't going to go away, and there are a dozen manufacturers all waiting eagerly to overcome some of its more irritating faults and failings.

Cop ing with cassettes

There doesn't, at first sight, seem to be much that you can do about the main limitations of cassette storage – slow LOADING and SAVEing and slow access to stored programs. However, if you check out the small ads in magazines like *Sinclair User* or *ZX Computing* you'll find several people advertising fast loader programs for cassettes that can double the speed of all your cassette operations. The price you pay for this is accuracy – so you had better make sure that you VERIFY every program! – but fast loading is possible without any fancy hardware at all. How? To answer *that* question I'll need to make a small detour into technicalities.

Baud with waiting

When you SAVE a program to tape the CPU feeds all the data it has stored about that program to the cassette output port, one byte at a time. If it did this at its own speed the result would be an enormous mass of electrical signals emerging from the port in a fraction of a second – too fast, in fact, to be successfully recorded in any intelligible way by your cassette recorder. So the flow of data has to be 'slowed down' by the computer. The speed at which data is sent out from one unit to another is known as the *baud rate* (after a Belgian telegraph engineer called Baudot), and a unit sending at 300 baud will transmit about 30 bytes in a second.

Now since the baud rate for sending and receiving data is controlled by the computer, you can actually persuade your Spectrum to double it by some cunning programming. This will halve the time needed for transfer of data to and from cassette, but it will also halve the reliability of the transfer. The rate was set at its present level for a good reason! So changing the baud rate doesn't guarantee a faster transfer. After all, if you have to do the job twice or three times it'll actually be *slower!*

Beefing up your cassette system

If you have trouble with your cassette storage, don't worry. You are not alone. Although the Spectrum is one of the most reliable micros for cassette storage, it and many others don't always work well with *every* cassette deck. The trouble is that cassette decks are really jumped-up dictation machines – they were never designed for data storage.

Your computer generates signals at a fairly high frequency – one, incidentally, that some very old machines have trouble coping with. New machines, on the other hand, tend to be looking for bottom notes that are hardly there, and amplify anything they can find to fill the gap. This can confuse your computer when you try to reLOAD the recording! Stereo heads can confuse it, too: sometimes they set up a pattern of interference on the very signals the computer is trying to read. The best bet is a cheap and unpretentious deck (but try it out first!) or a deck specially made for computer work (though they tend to be grossly overpriced for what they are). Even if you don't have trouble of this kind, you are pretty soon going to be very fed up with plugging and unplugging the lead from the EAR socket every time you LOAD and SAVE. But fear not – there are at least a dozen different manufacturers all falling over each other to sell you their particular solution to the tape deck blues.

The obvious place to start is with the cassette deck itself. Is it properly adjusted? If you're not sure, then **Hilderbay** can supply you with a useful alignment tape and manual that will allow you to check the *azimuth adjustment* on your recorder without using measuring instruments. In ordinary language that means you can find out if the record and playback head is properly aligned: if it isn't, buy the manual and read up what to do about it! In fact, the adjustment is fairly simple (see Fig. 23.1 for the general picture). Just make sure that the 'slot' at the front of the head runs precisely parallel to the tape. If this still seems a bit mysterious, you might like to consider a recorder specially 'tuned' to match ZX computers, also available from Hilderbay.

Next, what about the EAR lead? **Elinca** have the obvious answer, a simple three-way switch marked SAVE, LOAD and MAP. You leave the leads connected all the time: an LED reminds you of the setting and there is a meter and filtering unit to help you get the best possible results. When you aren't using the cassette deck you switch to AMP and use it as a BEEP amplifier – nice and simple and about £14.95. The CO-DER unit from **Jiles Electronics** has an amplifier and a filter to improve recordings, and a monitor that allows the computer to check that the program is being correctly recorded. If you have ever juked a cassette as unLOADable, you may regret it now! Another option is to use a sound and joystick interface such as the **Fuller Box**, which also has a built-in self-switching interface for cassette LOADING and SAVEing, and enhances incoming signals from the deck before they reach the computer. You'll find more about the Box and its capabilities in Chapter 24.

The **Stonechip** BEEP amplifier also includes a tape loading switch and a CUE button so you can use its built-in speaker as a microphone, and record program titles or other comments onto tape before downloading material from the computer. The sound it makes is very good, but the connections on mine were a bit loose and crashed several of my programs – you have to route the 9 volt supply from the power unit through the amplifier.

Finally, don't forget the range of power supply add-ons from **Kelwood**. Many of the variants in this range include tape LOAD and SAVE switches, as well as on/off switches for the entire system (another useful extra), and a neat storage

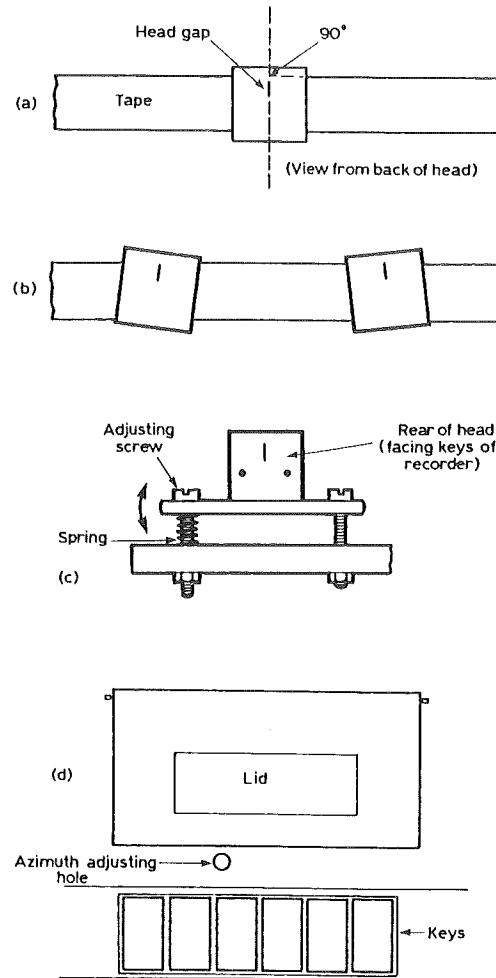


Fig. 23.1. Azimuth adjustment on an ordinary cassette deck. The narrow slit in the tape-head (a) is normally at 90° to the edge of the tape. This is the correct azimuth angle, but a surprising number of recorders have this maladjusted. Any deviation from this angle (b) causes muffled sound and poor loading. The angle can be altered by turning an adjusting screw (c) which is on the head mounting. This is often reached through a hole in the casing of the recorder (d). (Courtesy of Keith Dickson Publishing.)

place for awkward things like cabling, power units, and multiway mains adaptors. The power base range, which sits underneath the computer, doesn't (at the moment!) make provision for Interface 1. However, the larger mains unit, which arrives complete with a three-way mains adaptor, tape LOAD/SAVE switch, mains switch, speaker with volume control, and a compartment for the Sinclair power unit, is obviously OK for any system.

Network Computer Systems offer a 'multisave' unit that will allow you to save programs to two tape decks simultaneously. Ness Microsystems of Inverness can

supply a unit that actually controls up to two tape decks with BEEP commands! SAVE and LOAD switching is automatic (lovely!). The unit also includes an amplifier, and sells for £18.45 in kit form and £21.45 ready-built.

The more expensive units of this type are beginning to move towards a more useful kind of storage unit – one that is interactive with the computer, controlled by lines within the program itself. Wouldn't it be nice if your Spectrum could search an entire cassette tape 15 metres long in eight seconds, pick out the program, routine, or data that it needed with unerring precision, and LOAD it almost instantaneously – and all from your pre-programmed instructions? No need to just dream about it – it's possible, thanks to Sinclair's new *ZX Microdrive* storage system.

Memory expansion on a tape

The heart of the **Sinclair ZX Microdrive** system is a tiny tape cartridge 30 × 42 mm square and 5 mm thick. You could pack about ten of them into the average cassette box. Inside is a good grade of videotape cut into a strip half the width of cassette tape (1.9 mm to be exact) and joined into a continuous loop 15 metres long. Because the loop is continuous, and because it moves past the 'playback' head of the Microdrive unit at high speed, the Microdrive system is almost like an extra chunk of RAM. It allows you to store the memory-eating sections of a program – data, arrays, and even entire routines – in a readily accessible form that doesn't tie up the computer itself.

What amazes anyone used to cassette loading is its speed of operation. A typical BASIC program will LOAD in about seven or eight seconds. A machine code program could be even faster. And a SCREEN\$ will appear on screen in about four seconds. Given that you can store up to eighteen of them on one cartridge, and at least three in resident memory, you could produce animation sequences that way.

If £90 still seems a lot to pay for fast LOADING and SAVEing, remember that Interface 1 does a lot more than connect the Spectrum to the Microdrive unit. First, it contains a built-in RS232 interface (see Chapters 26 and 27) which allows you to hook up a printer directly. Secondly, it contains a networking interface (see Chapter 27 – for the moment it's enough to say that a pair of networked Spectrums can exchange massive programs at very high speed).

The Microdrive in close-up

The assembly for the Microdrive unit is as shown in Fig. 23.2, with Interface 1 mounted behind the Spectrum and the Microdrive unit attached to one side of Interface 1 by a ribbon cable. Do be careful how you connect this – it only fits one way! On the other side of the drive unit is a second socket that takes the special connector supplied in the Microdrive pack. This connector will link the unit to a second Microdrive.

A demonstration cartridge is supplied with the Microdrive. Switch on the

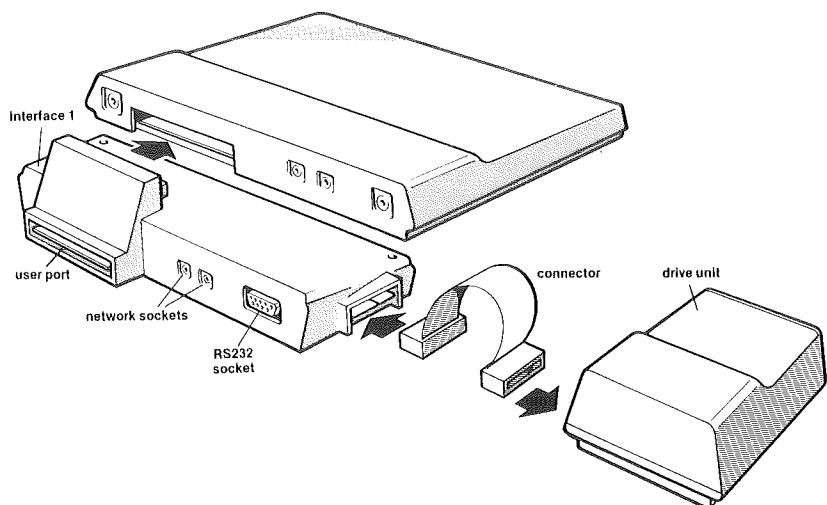


Fig. 23.2. Fitting Interface 1 and Microdrive.

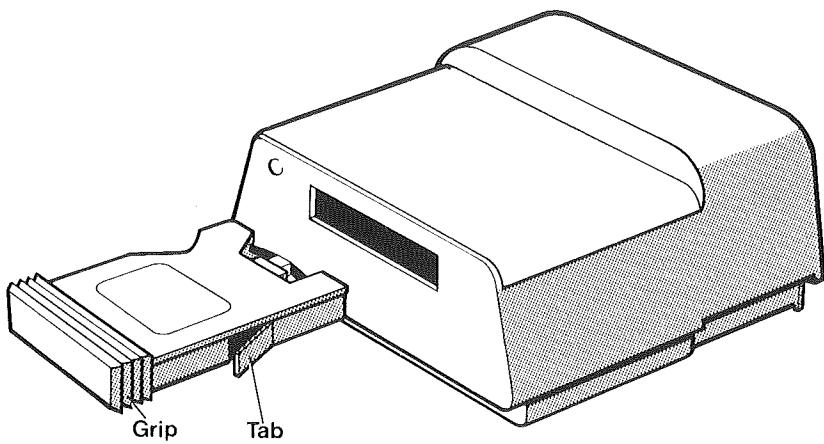


Fig. 23.3. Inserting a cartridge into the drive unit. The tab can be broken off rather like the tab on a cassette tape to prevent you from ERASEing a file accidentally.

system (this is important!) take it out of its case and slot it into the drive (Fig. 23.3). The plastic gripper at the bottom is *supposed* to come off, so don't worry if it does. Now key in RUN (the single keyword) and ENTER. The warning light on the Microdrive will come on, the drive will whirr entertainingly for a few seconds, and a menu will appear on screen. If nothing happens, or an error message appears, then you haven't loaded the demonstration cartridge. The demonstration cartridge and manual are themselves a good introduction to the Microdrive, but it

does help to know something about the system before you buy, especially as a whole new set of BASIC commands are available through Interface 1.

Speak gently to your Microdrive ...

If you've ever wondered what all those extra commands on the top row were for, this is when you find out. You'll also need to get to grips with some new ideas about your Spectrum and the way it works. Let's start with that top row, where the mystery keys, in order, are OPEN#, CLOSE#, MOVE, ERASE, CAT and FORMAT. For reasons that will become clear, I'll deal with these in reverse order.

When you buy a new blank cartridge (other than the demonstration cartridge) you are buying 15 m of blank tape. In order to use this tape, the Microdrive has to 'mark up' the tape into *sectors*, and this is done by keying in

10 FORMAT "m";1;"title"

'Title', of course, can be anything you want to call that particular cartridge – 'Games', or 'Addresses', or in my case 'Add-ons'! Each sector of a formatted tape can hold up to 512 bytes (in other words, half a kilobyte), and anything you store on a cartridge is stored in half-kilobyte chunks, even if the last section of what you're copying takes up less space than that. The FORMAT command, however, also 'checks out' the tape in the cartridge, and if a sector that is damaged or unusable is found, then it is effectively marked as unusable. One such section is the splice where the two ends of the tape loop are joined; others may be caused by careless handling or other damage. The speed of the tape past the heads is remarkably high (15 m in seven seconds must mean at least 2 m a second!) and sometimes it can buckle or crease – after all, it is only 1.9 mm wide! If reading this doesn't convince you that even Microdrive cartridges need back-up copies, then it should. Microdrive gives you a remarkably fast and effective storage medium, but it's not infallible.

Now that your tape is formatted, you're ready to store something on it. The BASIC commands for SAVEing to Microdrive or LOADing from it are very similar to the ones you use for ordinary cassette storage. However, each of them starts out with a variant on that jumble of letters, numbers and punctuation at the start of the FORMATting command. To SAVE a program, for instance, you would type in:

10 SAVE *"m";1;"program"

To LOAD a program from Microdrive you would enter:

10 LOAD *"m";1;"program"

To VERIFY, and to LOAD or SAVE machine code, screen strings (SCREEN\$) or data arrays you use the same format, with the extra details also used for cassette storage. So far these statements don't look very different from the ones you're used to, but there are two more you will not have come across before: CAT and ERASE. For instance,

```
10 CAT 1
```

will give you a list of up to 50 file titles that are stored on the cartridge in the drive at that moment – it even puts them in alphabetical order for you. No more wondering where you stored that vital set of data! ERASE is used rather like a storage command, in the form

```
10 ERASE "m";1;"program"
```

for instance. However, you don't have to add anything extra for machine code or data files. If you've stored a screen string called 'Fred' and you don't think Fred will be flattered by it, then ERASE "m";1;"Fred" will remove it for you.

Channelling the stream

I've left the three most difficult commands to last, and you won't be surprised to learn that understanding them means understanding the 'jumble of letters, numbers and punctuation' at the beginning of the other commands.

'M' in double quotes refers to the Microdrive. If you don't include it in commands like LOAD*"m";1;"program" then the computer simply won't accept your command. There are other letters that can also be used here, of which more in a moment. The number 1 is equally important – it tells the computer which drive unit you want to use. However, anything in double quotes in a command like this will be interpreted by your Spectrum as referring to a *channel*. Even if you don't have a Microdrive at the moment, there are already three channels working in your system: channel "k" (the keyboard channel), channel "s" (the screen channel), and channel "p" (the channel for the ZX Printer – which is present even if you don't happen to be using it).

On their own, these channels are useless. Until they can be connected there is no way for information to pass from, say, the keyboard to the screen. Normally the on-board programming will handle routine operations like this, but those without Microdrive might like to try the following experiment. Enter this short program:

```
10 PRINT "Channel K to channel
      S"
110 PRINT #2;"Channel K to chan
      nel S"
```

and RUN it. You will find that the computer treats both lines in exactly the same way, by printing the message to the screen. So what exactly is '#2'? In fact it's a *stream*, and the task of a stream is to form a link between channels. If that sounds a bit wet, try the next program.

```
30 PRINT #1;"Channel K to chan
      nel S"
40 PAUSE 0
```

and your message will appear in that previously unreachable space at the bottom of the screen. Line 40 is necessary to stop the computer's 'OK 30:1' message from blotting out what you've just printed. Now try this one:

```
50 PRINT #1;"Do you want to pr
      int this line again (Y/N)?"
60 PAUSE 0
70 IF INKEY$="Y" OR INKEY$="y"
      THEN GO TO 50
80 PRINT #0;"All right, be lik
      e that"
90 PAUSE 0
```

If you keep pressing the Y key you will eventually get an error message saying 'Out of screen 50:1', but you get the idea – here is a way of programming a menu without using the INPUT command. Used carefully, it can get you into your programs quickly and efficiently. Stream 3, incidentally, connects with the ZX Printer, so it will only work if you have the printer (or one like the Alphacom, which uses the same commands) actually connected. Using streams in your own programs can be very helpful – it means, for instance, that a simple number formula can replace a more complex choice between PRINT, LPRINT and INPUT. Try:

```
100 FOR j=0 TO 3
110 PRINT #j;"They seek me here
      , they seek me there"
120 NEXT j
130 PAUSE 0
```

You should finish up with the message at the top of your screen, two at the bottom, and another on your ZX or ZX-compatible printer.

Channels, streams, and Interface 1

The moment you fit Interface 1 to your Spectrum, things begin to get even more interesting. Streams 0 to 3 are already 'spoken for' by the computer, but with Interface 1 you gain access to two more channels ("m", a file on a Microdrive cartridge, and "n", another computer on a network) and *twelve* more streams, which are not spoken for at all. The value of all this depends pretty much on your own imagination. If you feel like some fun, try:

```
10 OPEN #4,"S"
20 IF INKEY$<>"" THEN PAUSE 3
30: PRINT #4;INKEY$;
30 GO TO 20
```

This has the curious effect of letting you type directly onto the screen. DELETE and CAPS LOCK won't work, but most other keys will, including the cursor key (though you can't actually see where the cursor is until you start typing!). What you have actually done is to open a stream to the TV screen and feed codes from the keyboard directly to it.

Streams and Microdrive

If you want to try something more useful, enter the program listed in Fig. 23.4 and RUN it. When entering room names, don't use more than 20 characters or the last few letters will be chopped off in the second part of the program. Try it now, before you read any further.

```

10 OPEN #4;"m";1;"rooms"
20 FOR j=1 TO 4
25 INPUT "Room number "+STR$ j
+" description?" r$
30 PRINT #4;r$
40 NEXT j
50 CLOSE #4
60 OPEN #5;"m";1;"rooms"
65 DIM r$(4,20)
70 FOR j=1 TO 4
80 INPUT #5;r$(j): PRINT r$(j)

90 NEXT j
100 CLOSE #5

```

Fig. 23.4. BASIC filing program using Microdrive.

So, what did it do? Let's take it step by step. In line 10 it linked stream 4 to a Microdrive file called "rooms". You probably heard the Microdrive whirr and then stop – actually it was looking for a file called "rooms" in case you'd already entered one. It was also setting up a clear space of 512 bytes *inside the computer itself* for storing the data coming in along stream 4. This clear space is called the *buffer*, and it's a term we'll be coming across again.

Lines 20 and 25 are straightforward enough, but what about line 30? Up to now you'll have been used to using the PRINT command to feed data to the screen – but this data isn't printed on the screen at all. Instead it's PRINTED along stream

4, and into that 512-byte buffer. And when j reaches the value 4, the stream is closed, in line 50. The command CLOSE #4 doesn't just 'switch off' the stream; it also empties the buffer – all the data in the buffer is transferred to the Microdrive cartridge. If you are PRINTing a lot of information to the buffer then this will sometimes happen during a program, before you CLOSE the stream, when the buffer has received a full 512 bytes. You'll hear the Microdrive whirr into action to deal with it.

The next part of the program reads back what you've put onto the Microdrive file, only this time it stores it in a data array. The DIMensions of this array are set in line 65. Line 60 sets up another buffer, but because the file "rooms" already exists on the Microdrive cartridge the results are rather different. This time the contents of the Microdrive file are loaded into the buffer. Incidentally, if you tried to PRINT anything to stream 5 after the OPEN # command in line 60 you'd get an error message saying 'Writing to a "read" file'. You can't add anything directly to a file stored on Microdrive. If you want to change it you'll have to move its contents into the computer's on-board memory, alter them there, and then store them back on Microdrive – and if you want to keep the original file on the same cartridge, you'll have to give your altered file a different name.

Line 80 does a familiar job in an unfamiliar way. Instead of taking INPUT from the keyboard, in the way you're probably used to, it takes it from the buffer. The effect is much the same, though, and in this case each string read by line 80 is given a number in the r\$ array. Once the program has finished, and stream 5 is closed, the computer will have an array of four strings stored in RAM. To check, try entering PRINT r\$(1) as a direct command, and the first entry you made should appear on screen.

There's one final command we need to look at. If you entered the Microdrive program for 'rooms' earlier on, just key in, as a direct command –

MOVE "m";1;"rooms" TO #2

and press ENTER. TO should be entered using SYMBOL SHIFTed F, of course. You should now see the contents of the 'rooms' file appear on screen – and yet any program you had in the computer remains totally unaffected. MOVE has even OPENed and CLOSEd the file for you. This is a very useful trick – you can even use it to send a data file to your ZX or ZX compatible printer by replacing #2 with #3. If you can get hold of two Microdrive units and a ZX-type printer you might like to try:

MOVE "m";1;"rooms" TO "m";2;"rooms": MOVE "m";2;"rooms" TO #3

The results are quite entertaining; and the whole thing happens without a screen display, and without affecting any program that happens to be in the computer at the time. Notice that files on different drives can have the same name. You can actually MOVE files to a different place on the same cartridge, but in that case you'll have to give the copy file a new name to avoid confusing your poor innocent little Microdrive. This will only work with files that have been PRINTed to Microdrive using the OPEN # and CLOSE # commands, but you can always

convert ordinary data files to this format. If you had a file called 'load' containing 35 strings of characters you could enter a program like:

```

10 LOAD *"m";1;"load" DATA 1$(
)
20 OPEN #4;"m";1;"data"
30 FOR j=1 TO 35
40 PRINT #4;1$(j)
50 NEXT j
60 CLOSE #4
70 MOVE "m";1;"data" TO #3

```

and line 70 would even print out the contents of the new file on the ZX printer for you!

You can use streams to make many other useful connections, too. If you're interested, glance ahead to Chapter 27. For the moment, it's enough to understand the general principles. Believe it or not, those same principles are used by many of the disk drive systems now available for the Spectrum – and some even use modified versions of Microdrive commands.

So who needs disk drives?

Microdrive is an excellent system for the ordinary user. It's cheap, flexible, and fast, and it gets over most of the major problems of cassette storage as well as giving you a whole host of new features. However –

- It doesn't identify file *types* with the CAT command – only file names. If you've forgotten whether 'Fred' is a BASIC program, a string array, a number array or a machine code routine, then hard luck.
- It is possible to wipe a cartridge accidentally.
- Many of the most useful commands will not work with files loaded using conventional LOAD operations.
- At £4.95 a time the cartridges are very expensive.

On several occasions I have been trying to use Microdrive and finished up with a locked-up computer and a continuously purring drive unit. Obviously one can't remove the cartridge in this situation, so the only solution is to switch off the power – and risk losing everything on the cartridge. Sinclair themselves are aware of these problems, and the Interface 1 ROM is currently being revised, but I have found the system so useful as it stands that I don't regret buying it. However, it does make me look very carefully at any more expensive storage system. Given that you're willing to spend the money that most disk-systems are going to cost you, what are you getting for your money? The answers should be:

- Speed (and you thought Microdrive was fast!)
- Greater storage capacity.

- Greater flexibility in use.
- Proven reliability.

Disk drives – what they do and how they do it

The difference between a disk drive and Microdrive, broadly speaking, is the difference between a cassette tape and a gramophone record. If you want to find a particular piece of music on a cassette, you will have to wind the tape through until you get to it. If you want to find it on a record, you can usually do it by picking up the stylus and lowering it again in the right place.

A disk storage system uses a *read-write* head instead of a stylus and a flexible mylar disk coated with magnetic oxide instead of your favourite album. Unlike your favourite album, the disk is kept in a protective sleeve all the time, and this sleeve cleans it as it rotates. The system can find a given file rather faster and more accurately than you can find a track on the album because part of any disk is reserved for the *directory*, a detailed list of what's on the disk and where it is. To map out the disk the system divides it into *sectors* (rather like the slices of a cake) and *tracks*, a series of concentric rings on the disk. The sectors are marked by holes punched in the disk. Everything you put onto the disk is assigned one or more sectors of a given track. So, when you want anything, the system simply consults the directory and lines up on the right part of the disk almost immediately.

As long as the system is reasonably efficient, this means that you can get to files on the disk either *randomly*, as you need them, or *sequentially*, in a fixed order determined either by the files themselves or by programming. True random access is something you don't get even with Microdrive. Ideally, you can achieve this control with a few short, simple commands, perhaps similar to the BASIC ones you already know. With a really good disk drive system you almost seem to be operating a 200K RAM computer! However, disk drives are subject to the same bugs and jitters as any mechanical device anywhere, so don't get carried away. You need to look at your prospective new system with a cold and critical eye.

Disk drives – what you see and what you get

There are currently four Spectrum disk systems, either available or at working prototype stage, and a fifth in development at what is expected to be a highly competitive price. You would be very well advised to check the availability and price of any system that interests you before placing an order. You'd be even better advised to try it out for yourself. To get you started, here are a few pertinent questions.

- Where does the operating system reside?
- How much on-board memory does it use?
- What sort of drives does it use? Could I use a different kind if I wanted to?
- Does the price include the disk drive, interface, cabling, and power supply?
- Does it include any disk software?

- What other software can I use?
- How efficient is it? Can it handle double-density and double-sided disks?

The operating system

The idea of an operating system (sometimes called DOS, for Disk Operating System), may be new to you. In fact your computer wouldn't work without an operating system, but because you can't do anything to it you tend not to notice it. The operating system is the built-in program that directs and co-ordinates all the various activities of the MPU and peripherals. Microdrive has its own operating system built into Interface 1; disk drives need one, too, and theirs can't be built into the computer itself. Obviously a complex drive set-up could eat up a lot of your 48K this way. Find out how much working space you have when the system's up and running – and make comparisons with other systems.

Software

Nowadays a good system should include some software in the basic package. A typical selection would include a word processing program, a filing program of some sort, and a 'spreadsheet' program for accounting (useful if you're going into business for yourself but not so useful otherwise). You should be able to RUN ordinary BASIC programs without any trouble, provided that they don't use more RAM than the DOS has left you, but check with the supplier if you want to buy large machine code programs, especially programs like assemblers that use high memory (i.e. addresses above 60000).

Choosing your drive unit

Listed below are the three main standards for disk drives at the moment.

- 5½ inch (just about the industry standard).
- 3½ inch (Sony's attempt to improve on this).
- 3 inch (Hitachi's alternative – cheaper at the moment).

(Others are available, but I wouldn't recommend them.) The size refers, of course, to the diameter of the disk. Equally obviously, a smaller disk needs a smaller drive unit, which means less wasted space for you. Many systems offer 5½ inch as standard. Others have gone for 3 inch. Both systems offer good value for money. Typically, a 40-track single-sided single-density 5½ inch floppy disk can accommodate up to about 200K – twice the capacity of a Microdrive cartridge – and will cost you about £1.70. This is about the smallest capacity available. 400K, or 800K double-density double-sided disks cost only about £3. However, it's not quite that simple.

Getting the most out of your disk

Efficient use of the disk really depends on how well the system as a whole – disk drive, operating system, interface and computer – actually performs. A good

system can pack a disk with information, but a poor one may never let you reach its full potential. Most Spectrum disk systems will handle 40-track single-density disks. Drive units for these in 5½ inch format come at about £150, which doesn't look too bad. However, you can expect to pay very nearly the same again for the interface and operating system.

Among disk systems look out for the new Byte Drive system from ITL Kathmill, which is intended to have *two* different operating systems on the system disk. One will be a specially written 'Spectrum DOS' and the other will be an implementation of the CP/M standard used for a great deal of very useful business software, and some very exciting games, too! Another system looking good at prototype stage is the Morex, with extremely user-friendly commands and some useful software features. Available now is the FIZ interface from Primordial Peripherals – good, user-friendly manual but a bit inefficient compared to some of the new arrivals. Finally, there's a completely new system from Radofin Electronics at Hyde House, London NW9 6LG (Tel: 01 205 0044). It's a complete package of drive, interface, power supply and cables for an anticipated price of about £130.00, using 3-inch disks formatted *sequentially* like a gramophone record. Capacity each side is 72K unformatted, 51.5K formatted, and a complete disk can be read and written in eight seconds a side. For full details contact Radofin – production models were not available at the time of writing.

And now a word from your computer...

Chapter Twenty-four

Speech, Sound and Music

There you are, sitting peacefully down to an evening's computing. Without looking at the screen you key in the LOAD command – and a Dalek-like voice suddenly says 'Load! Enter! Quote! Quote!' You are just wondering if micro madness has finally set in when you see the small black box sitting at the back of your Spectrum. Experimentally, you press other keys. It responds by telling you whatever it is you've just entered – anything from 'comma' to 'randomise', though for some reason it balks at the tilde (~). This is the **Currah Microspeech**, one of the best and most easily programmable speech synthesisers for the ZX Spectrum. The key readout is optional – if you get fed up with it just tell your new friend to

LET keys=0

and you will just hear judiciously amplified BEEPs – another useful feature. You can restore it with the command

LET keys=1

These commands can also be used in programs, of course.

So talk to me!

If you really want your computer to talk to you, there is quite a wide choice of speech units on the market. However, the shortlist boils down to about four or five, and of those three actually use the same chip, which means they can be programmed in more or less the same way. Points you need to look for are:

- Sound: can you actually understand the words?
- Versatility: how much can it actually do?
- Programming: how hard is it to put words together, and how much help does the software give you?
- Connections: can you use the unit at the same time as other add-ons?

Price isn't really a factor, except with the DCP S-pack and the William Stuart, as most speech units cost about £30. To get a clearer picture of how the available units compare, it'll help to take a brief look at the way they work.

Saying hallo to the allophones

The Currah, like many other speech synthesisers, uses *allophones*. Allophones are

electronic copies of the sounds we use in everyday speech – and you can ‘simulate’ English with just 64 different sounds. If that seems incredible, remember that you can write every word in the English language using just 26 letters: it’s not the number of sounds that’s important, it’s the number of possible combinations. Once you get the hang of it, programming allophone speech units isn’t as difficult as it might appear, though it helps if the software is good. But getting really good sound is quite difficult. Obviously it would be a lot easier if the actual selection of sounds was done for you, using built-in firmware that manipulated the chip on your behalf. This is the principle behind the DCP S-pack, and the result is the best speech synthesis you are ever likely to hear. But the choice of words available from a unit like this one is comparatively small – you can’t assemble new combinations yourself – and you have to rely on the manufacturer to give you the words you want to use.

Choosing your unit

Most of the speech units currently on the market (apart, of course, from the DCP S-pack) use the same chip, but each of them implements it in a slightly different way. One thing most of them have in common is their use of the OUT command, one you haven’t yet come across in this book. In effect it acts rather like a POKE, but the piece of memory it’s POKEing is outside the computer! The manuals always tell you which number to use for this command – in the case of the Cheetah, for instance, it’s OUT 31 if you have Microdrive and OUT 7 if you don’t. The

```

10 PRINT "SPEECH SYNTHESISER E
XAMPLE"
20 INPUT "Key in the OUT numbe
r used by your unit (see text
) ";out
30 INPUT "Key in the length of
the pause between allophones
(1 to 10) ";pause
40 IF pause<1 OR pause>10 THE
N GO TO 30
50 FOR j=1 TO 18: READ s: OUT
out,s: PAUSE pause: NEXT j
60 DATA 61,39,19,13,12,44,55,0
,0,33,60,0,0,14,19,33,51,0
70 INPUT "Was that OK? (Y/N) "
;a$: IF a$<>"Y" AND a$<>"y" THE
N CLS : RESTORE : GO TO 10
80 PRINT "'OUT port is ";out'
"Pause is ";pause

```

Fig. 24.1. Demonstration program for speech units using the same allophone system as the Cheetah.

numbers that follow the OUT command and number call up individual allophones from the speech chip. You can see how it works from the program listed in Fig. 24.1. This will work with most speech synthesisers using this form of programming, including the Cheetah, the Fuller Orator, and the William Stuart Chatterbox. Of these, the Orator can be fitted into the impressive Fuller Box, which also includes a sound synthesiser, joystick port, and cassette interface; the rather pricey William Stuart can form part of a remarkable speech and sound system (see below); and the Cheetah stands alone but beats the others on ease of use and good documentation.

The Currah Microspeech is programmed in a completely different way, using actual keyboard letters. For instance, if you type in

LET SS="ha(l)(oo)"

your Currah will give you a cheery robotic greeting. Documentation for this unit is excellent – it’s easy to use, and has become popular with many software manufacturers, who program an option for it into many games. The sound comes out through your TV speaker (so it’s controllable!) but this does cause a minor snag. The Currah’s signal interferes with the TV picture, and this interference even carries over into the TV speaker. You can cut it to a minimum by using the tuning screw on top of the unit, but I found it very irritating, especially over long periods of use.

Using speech synthesisers

Using units like these purely for games may seem rather frivolous, but the Currah in particular is almost designed to be frivolous. They can be particularly useful, for instance, in confirming legal commands entered during an adventure game! For the serious minded reader, a similar piece of coding in a serious business program could be a handy check, ensuring that the user *knows* a valid command has been safely entered before going on to the next stage. A unit like the Currah is particularly useful because it confirms each individual keystroke. It’s too slow to keep up with a touch typist, but a non-typist who needs to look at the keys a lot might find it comforting to hear the ‘right’ command being entered, and useful to hear if the *wrong* key has been pressed, too.

Talking back

So far we’ve been holding rather one-sided conversations with the Spectrum, limited to what you can program into one or other of the speech synthesis units currently on the market. But suppose *you* could talk to your Spectrum, and it could ‘understand’ what you said?

Well, it’s possible, up to a point. Getting a computer to recognise speech is not too easy at the best of times, and even if it does it’s only going to recognise the voice of the person who programmed it. Units like this work by analysing the sounds you make and converting them into numbers. The gadget that does it is called an

analogue to digital converter, which basically means anything that turns non-number information such as voltage variations into numbers. Typically a unit will ask you to say the same word several times, average out its readings each time, and come up with a formula that will allow it to recognise the same set of sounds next time around. Not surprisingly, you can usually fool it with similar sounding words (like 'wine' and 'dine'), and if your voice happens to be a bit off after a night of wining and dining your faithful computer may totally ignore you. (Who said computers aren't like people?) To program any speech recognition unit you need to work in a quiet room, and speak very clearly and consistently. That takes a little practice, so be patient. There are two systems currently available: Big Ears, from William Stuart, and Micro Command from Orion Data.

Using speech recognition systems

The main problem with systems of this kind is the time any unit takes to recognise a word and respond to it. The answer is to structure your programs so that there are only a few words to choose from at a time, preferably as different from each other as possible. If you're using the unit to choose options from a menu you'll find that this sort of structure makes sense in any case. You should also allow for Murphy's Law and include an option that allows the user to start again, or return to the previous menu – misunderstandings are fairly frequent!

Perhaps the most exciting application for this type of unit is to team it up with some of the control devices discussed in Chapter 29. I used it with the REL 4.2 unit from Harley Systems and spent an entertaining hour or so switching lights on and off with pure voice power. This is the perfect microcomputer application for the incurably lazy, like me, and for people with handicaps who might have rather more of a problem operating a conventional switch. Using the William Stuart system the speech unit could confirm that your command has been understood!

Three-part harmony

It is entirely possible that the idea of a talking computer leaves you cold. Musical computers, on the other hand, are almost commonplace, and it will come as no surprise to learn that there is a wide choice of sound and music synthesisers for the Spectrum. As with the speech units, many of them are based around the same chip, and can be programmed in almost exactly the same way. The changes rung by different manufacturers range from a bare-circuit board model at £19.95 to the top-flight version of the Fuller Box, the Master Unit, which weighs in at £54.95 and includes the Fuller Orator, a joystick interface, and an electronic self-switching cassette interface. As with the speech synthesisers, what you're really looking for is

- Good sound quality.
- Easy programming.
- Compatibility with other units.
- Value for money.

Value for money really becomes an issue with these units – the price range is wide, and depending on your own abilities the best buy won't always be the expensive unit in the plush box. Software's important, too – it isn't fair to expect the new user to understand and handle the 14 different variables that normally have to be programmed! However, since the best sounding units are not always the best documented, let's take a look at a program that will work with most of the units currently available for the Spectrum.

Singing the registers

The full listing for 'The Sound Program' is given in Fig. 24.2: and as you can see, most of it is actually taken up with the text needed to make it easy to use! Enter it properly and you'll get a screen display like the one in Fig. 24.3. The first figure in each line is the *register number*; the next figure shows the current value of what's in there (all yours will start at 0), and the text tells you what it's for. There are

```

10 PAPER 1: INK 7: BORDER 1: C
LS : PRINT "THE SOUND PROGRAM"
20 INPUT "Enter the OUT number
your unit uses to select a re
gister." 'outreg
30 INPUT "Enter the OUT number
your unit uses to send inform
ation to the registers." 'outval
40 CLS : LET temp=0: LET place
=0: DIM r(14): DIM s(14): DIM m
(14)
45 FOR j=1 TO 14: READ m(j): N
EXT j
47 DATA 255,15,255,15,255,15,3
1,255,16,16,16,255,255,15
50 RESTORE 60: FOR j=0 TO 13:
READ r$: PRINT AT j,0;j: PRINT
AT j,3;r(j+1): PRINT AT j,9;r$:
NEXT j
60 DATA "A channel fine tune"
70 DATA "A channel coarse tune
"
80 DATA "B channel fine tune"
90 DATA "B channel coarse tune
"
100 DATA "C channel fine tune"
110 DATA "C channel coarse tune
"
120 DATA "Noise control"
130 DATA "Noise/tone selector"

```

```

140 DATA "A volume"
150 DATA "B volume"
160 DATA "C volume"
170 DATA "Waveform fine tune"
180 DATA "Waveform coarse tune"
190 DATA "Select waveform"
200 PRINT AT place,3; FLASH 1;t
emp; "
210 IF INKEY$="6" AND place<13
THEN LET place=place+1: PRINT
AT place-1,3;r(place);": LE
T temp=0
220 IF INKEY$="7" AND place>0 T
HEN LET place=place-1: PRINT A
T place+1,3;r(place+2);": L
ET temp=0
230 IF INKEY$="5" AND temp>0 TH
EN LET temp=temp-1
240 IF INKEY$="8" AND temp<=m(place+1) THEN LET temp=temp+1
250 IF INKEY$="0" THEN OUT out
reg,place: OUT outval,temp: LET
r(place+1)=temp
260 IF r(8)=255 THEN DIM r(14)
: CLS : GO TO 50
270 GO TO 200

```

Fig. 24.2. Sound unit demonstration program.

| | | |
|----|---|-----------------------|
| 0 | 0 | A channel fine tune |
| 1 | 0 | A channel coarse tune |
| 2 | 0 | B channel fine tune |
| 3 | 0 | B channel coarse tune |
| 4 | 0 | C channel fine tune |
| 5 | 0 | C channel coarse tune |
| 6 | 0 | Noise control |
| 7 | 0 | Noise/tone selector |
| 8 | 0 | A volume |
| 9 | 0 | B volume |
| 10 | 0 | C volume |
| 11 | 0 | Waveform fine tune |
| 12 | 0 | Waveform coarse tune |
| 13 | 0 | Select waveform |

Fig. 24.3. Screen display from sound unit demonstration.

fourteen registers, and as far as you're concerned they simply hold numbers that you can enter by using the cursor keys. Move up and down to find the one you want, move left to reduce the number or right to increase it, and press \emptyset to enter the new number. If you have a joystick, try adapting the program to work with it.

What I've called 'channels' A, B, and C control the *pitch* of the sound, i.e. whether it's high and thin or low and growly. You can change the pitch by altering registers 0 to 5, and the volume by altering registers 8 to 10, so when it's 0 you won't hear anything! However, if you set one of these registers to 16 then it comes under 'waveform' control. This will alter the 'shape' of the sound, and you can program it using registers 11 to 13. Odd numbers in register 13 will give a short sound followed either by silence or a sustained tone. Even numbers give a regular pattern of sound. Registers 11 and 12 control the variation of the sound – if you like, the size of the wave!

Yes, I've left two out. Register 6 lets you experiment with the noise generator – to get the most out of it you need to set register 7 fairly carefully. The only way to appreciate what these registers do is to switch the channels to noise (rather than tone, which makes *nice* noises) and try entering different values into register 6.

Have fun!

Making your choice

The choice of available units is large, but one of the cheapest, the uncased Add-On unit offers excellent value with both analogue and switch joystick interfaces and good programming. Also well worth looking out for are the excellently documented Petron and Signpoint units. The second unit for the Fuller Box is good, but badly documented. The William Stuart unit is, too, but offers the chance of stereo sound generation through your domestic hi-fi. Like all William Stuart products, it's rather expensive, especially as it has no case, though it does feature an input/output port. The Timedata ZXM is a useful unit, slightly let down by its software, but offering a handy joystick port. Bear the points listed above firmly in mind and you shouldn't have too much trouble making a sound choice.

It's been a long day, and it's time to try out some other Spectrum special features. Did you know that you'd bought a quite reasonable graphics generator?

Chapter Twenty-five

The Computerised Drawing Board

Are you a frustrated artist? Have you ever felt that you might have a talent for drawing and design if only someone would give you a chance? Or, perhaps more likely, have you ever felt that you could do a better job on graphics than most of the games designers? If so, then the good news is that your computer can now be fitted with some powerful graphics hardware that will make the task of any microchip Michelangelo that much easier. And it can all be done without breaking the bank.

The graphics file

For the Sinclair Spectrum there are three different types of graphics hardware currently available:

- Lightpens.
- Digital tracers.
- Graphics pads.

A *lightpen*, at least in an ideal world, is a handy little gadget that allows you to 'draw on the screen' with a device very similar to a pen. As well as drawing pretty pictures you can also use them to 'point out' items on a screen menu – especially useful for business applications or dotty adventure games. The *digital tracer* is a delightful device that looks rather like the 'pantographs' I used to buy to help me copy drawings in a larger or smaller size. And effectively it does just that, allowing you to copy flat artwork of any size (within reason) directly onto the computer. You can also create original artwork directly, or from your own rough drawings. The *graphics pad* is rather like a combination lightpen and digital tracer, but the pen, instead of operating on the screen, is used on a sheet of glass or perspex mounted over a special sensing surface. As the pen is pressed down, the tip is pushed back into the barrel and operates a microswitch. The pen 'reads' a crosswire grid rather like a super-fine version of the grid underneath the Spectrum keyboard. The result is a screen image that is an extremely accurate copy of the movements you have made across the pad.

Introducing the lightpen

To understand how a lightpen works you need to understand how a TV picture is actually built up on screen. The 'picture' is actually no such thing – it's the trace left

by a dot of light (or three dots in the case of a colour TV) tracking backwards and forwards across the screen so fast that it manages a complete scan in one twenty-fifth of a second. That's fast, but the Spectrum is faster – in that time it could carry out up to 140,000 operations! The lightpen acts as a sort of electronic stopwatch – it can detect where it is on the screen by checking how long it takes the scanning dot to reach it! If its response is accurate (and that will depend on how good its software is as much as anything else) the result is a pair of x and y co-ordinates that you can use as you wish – to fix a point on screen in the memory, to draw an INK pixel there, or to check which display line the pen is pointing at so that you can send your program to the chosen subroutine. If all this sounds complicated, it is – but not for you. If you're programming in BASIC for a lightpen, you're usually dealing with a single number that can be treated just like any other variable.

However, if a lightpen is to do any of these jobs it must be

- accurate – stray light can affect the reading;
- supported with fast and well written software;
- easy to use in your own BASIC programs.

To make sure the pen is accurate, the software will normally include a *calibration routine*. This checks that the reading going back to the computer agrees with what you can see. If it doesn't, then the neat black line you are trying to draw could appear three inches up on the left-hand side. Calibration is important! Best all-round buy is probably the dk'tronics Mk. 4, though the Custom Cables International pen offers more limited options at a lower price.

A trace of artistry

The lightpen is a useful graphics tool and the dk'tronics model can be a very useful

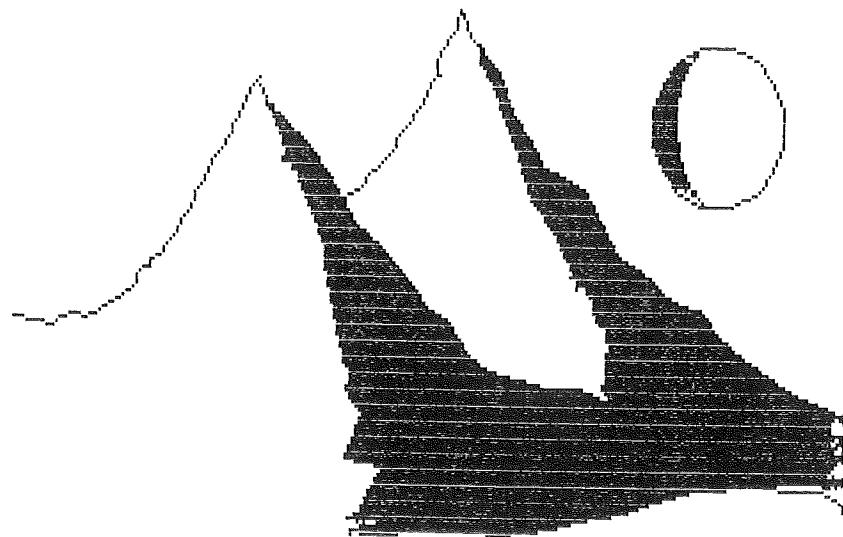


Fig. 25.1. Artwork produced in a few minutes using the RD Digital Tracer.

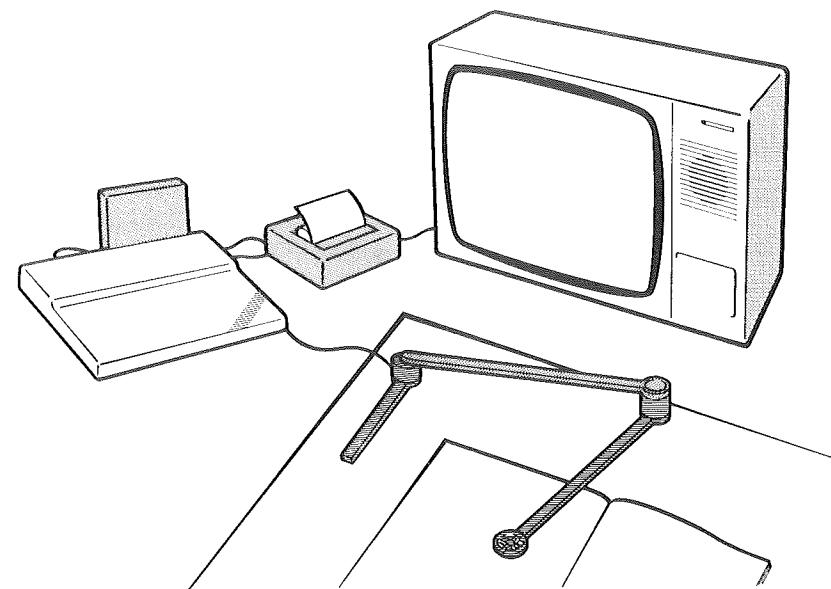


Fig. 25.2. The RD Digital Tracer in use.

menu selection device, but one thing it can't do is to copy images directly from flat artwork. If you want to create detailed graphic images using the full resolution of the Spectrum, you'll need something else, and the cheapest available alternative is the **RD Digital Tracer**. Figure 25.1 is an example of what it can do, and the way it's arranged for use is shown in Fig. 25.2. The principle behind it is comparatively simple: there are only two moving parts, and both contain a potentiometer (which, you will remember, is also used in some joysticks). If you've bought the tracer and you want to see the effect of the two potentiometers for yourself, then try the following program with the tracer connected:

```
10 OUT 31,1: PRINT AT 12,0; IN 31; ""
20 OUT 31,0: PRINT AT 14,0; IN 31; ""
30 GO TO 10
```

You'll get a printout of the readings they are producing on screen. The software converts these readings into x and y co-ordinates.

The software is supplied on a cassette tape – one side has the 48K version of the programs and the other has a 16K version with a few alterations to make allowances for the smaller RAM. Each side includes four routines – a basic 'draw' program, a 'scaling' program, a 'retrace' program that lets you copy a shape onto other parts of the screen, and a routine for creating user-defined graphics. A fifth program combines all the other four. The tracer is simple to set up, and simple to use. My sample was let down a little by its software, but RD tell me that a new cassette is on the way, and they are working on an improved version of the tracer itself.

Besides allowing you to trace outlines, the software includes some simple

routines for filling, colouring, and deleting, though the fill routine in particular was rather difficult to use on my sample. In fairness, this was probably because it makes a brave attempt to prevent 'leakage' through gaps left accidentally in outlines – after all, there's nothing more frustrating than watching ink spilling out into an area you meant to be paper colour, and wiping out any line detail you might have left there into the bargain.

Tracing outlines accurately takes a little getting used to – if you move too fast the outline on screen tends to 'cut corners' because the software can't absorb all the co-ordinates you're plotting quickly enough. The other problem is shaky hands! The tracer is remarkably sensitive, and even small movements tend to be magnified on screen! The answer is to use the scaling program supplied by RD and copy a fairly large image – ideally a good bit larger than A4. The reduced image that appears on screen should be a fairly good likeness if you're reasonably careful – and if you're not, the program allows you to erase small areas quite easily.

The tracer arrives complete with a template for setting it up and a reference sheet giving you very useful information like screen co-ordinates and memory areas for every part of the display.

Once you've started to explore the tracer software you'll soon begin to see other possible applications. For instance, suppose you replaced the PLOT commands with a series of adventure game subroutines called by different values of x and y?

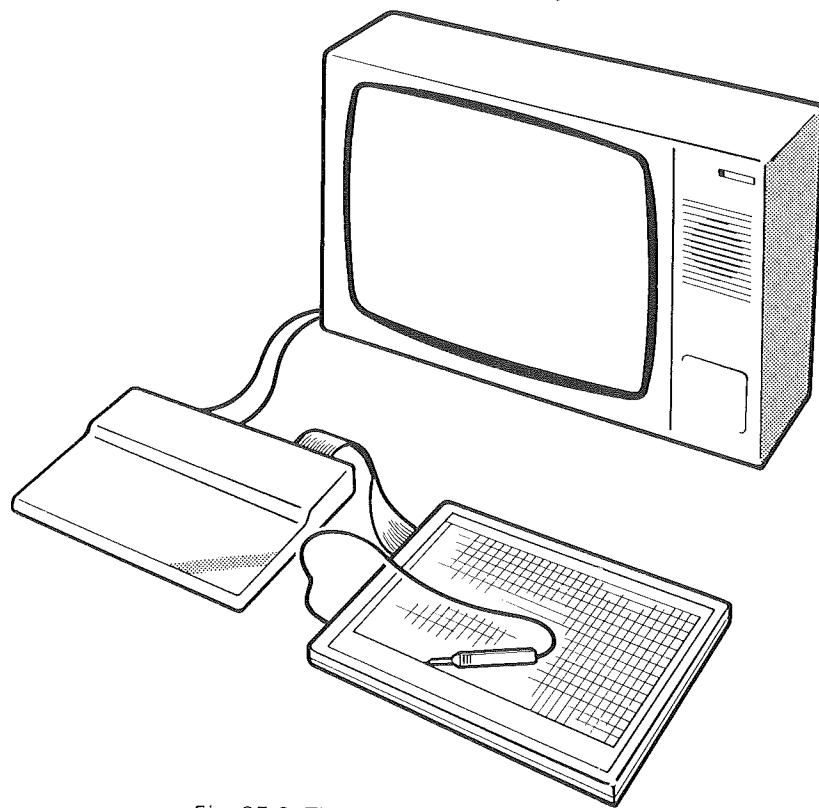


Fig. 25.3. The British Micro Grafpad.

That way you could lay a map – an ordinary Ordnance Survey map say – under the tracer, and do a complete orienteering exercise! I can see it now. 'You are on the edge of a high cliff. The mist is coming down. What do you do?' Quite seriously, this could be a valuable educational exercise – and you could line up the tracer on the map very accurately by using a modified version of the scaling routine, which lets you plot precise values for the corners of a document into the computer's memory.

As you will have gathered, I find the RD Digital Tracer extremely hard to resist. If you're in the least bit serious about graphics it will be an invaluable addition to your armoury. The promised improvements should make it an even better deal – but a tool like this for around £55.50 (or £75.50 for the 'professional' version, which can cope with larger artwork) is a bargain by any standards.



Fig. 25.4. High resolution artwork produced on the Grafpad.

Grafpad – the computerised sketchpad

Last, but by no means least, is the **British Micro Grafpad** (Fig. 25.3). I have placed it late in the chapter because of its asking price, currently £143.75. Why, you might ask, should you pay the price of your computer for a peripheral? The answer's very simple – because buying Grafpad is like buying a completely new computer. The unit arrives complete with a full manual, all necessary software, the Grafpad, the operating pen, and all necessary connections for your Spectrum. When I tried it out at British Micro in Watford it was connected to a Cub Monitor (see Chapter 28) and the effect was staggering. The accuracy and detail that are possible with this system make using it a pure joy (see Fig. 25.4). My main problems were getting to grips with the new keyboard (the kit includes a keyboard overlay to help you work out the dozens of new functions made possible by the software) and some initial uncertainty with the pen itself.

The pen contains a microswitch so that it will only operate when you press gently down on it. What happens next depends on which of several possible modes you are in. You can use the pen to define points for geometrical shapes – in this case moving it around the Grafpad will produce a shape that varies according to the pen position. When you get one you like, a keypress will fix it on the screen display and in memory. Or you can use it freehand – a fast, flowing stroke is best in this case, as the pen is slightly sensitive to jittery hands. Switch to another mode and the pen becomes a paintbrush, accurately filling in selected areas with your chosen ink colour. In yet another mode it's an eraser – and because it's a pen it's easy to be accurate with it.

British Micro software support for the Grafpad is excellent. Already available is a computer-aided design program that allows you to create and store a battery of user-defined shapes, take them from memory, and place them anywhere on the screen. The program cost about £20 and, if you want it, it's worth it. The possible applications are almost limitless – anything from designing electronic circuitry to drawing up detailed architectural sketches. I also saw a fascinating perspective program under development – just the thing for people like me, who never seem to get perspective right without outside assistance.

To go into greater detail about the Grafpad would only confuse the issue – it takes time and practice to familiarise yourself with its many facilities, and the best way to see what it can do is to watch it being demonstrated by someone who has already mastered it. Fortunately it has been making many exhibition appearances recently, and I have no doubt it will make many more. Go and see it for yourself.

The Grafpad is not a toy. It's a very clever and carefully designed concoction of hardware and software that will be a boon to professional designers and technical artists, and pure joy to the talented amateur. If your talents don't really lie in this direction, forget it – you could finish up frustrated and disillusioned. However, if you've been looking for a really effective way to create first rate graphic designs on the Spectrum, and you have the necessary ability, this is the tool you have been waiting for.

Getting it off the screen...

Of course, even the best graphic designs are not a lot of use in the real world unless you can get them off the monitor screen and onto paper. Fortunately that's a lot less difficult than it sounds – in fact there are several computer printers now on the market that will give you full-colour printouts. It's time to look at just what is available on the computer printer scene.

Chapter Twenty-six

Getting it in Print

by Phil Gardner

Getting it on paper...

Way back in the dear dead days when men were men and ZX81s were computers I can remember copying down screen LISTings of my painfully entered programs by hand as I tried to work out why subroutine 3000 was constantly giving me an error message whenever I jumped to it from 290... Yes, I could have scrolled backwards and forwards all the time, but the keys were beginning to get tired and so was I. The fact is that there's no substitute for 'hard copy' – a listing on paper which you can check, recheck, and write rude messages on. And while pen and paper may have been OK for William Shakespeare, BASIC spelling and grammar are a lot less flexible than English, and computers don't take kindly to mistakes. This is probably how many people start thinking about printers, and once you've actually started to use one for your program listings you'll never want to use anything else. And listings are only the beginning. Plenty of programs produce displays and information that are lost as soon as they reach the next CLS command – unless, of course, your printer has made a hard copy for you first.

Maybe you're a games player – and maybe you don't think a printer would be much use to you. But imagine playing a game like *The Hobbit* and being able to keep a printed record of what you'd done and the appalling things that had happened to you as a result. At least, next time you would have a way of knowing what *not* to do. With a good printer you could even run off a copy of the better screen displays – or your own masterpiece, hot off your RD Digital Tracer or your lightpen (see Chapter 25)!

Making the choice

There are so many printers on the market at the moment that you may well find it rather hard to sort out what you want and what the best one is for your particular needs. The good news is that the prices – which used to be geared to business users – have fallen rapidly in the last year or two and new models, with more and better facilities, are appearing all the time. This is wonderful, but it makes the choice even more difficult! Still, if you just want it for program listings, the look of what it produces really isn't important (as long as you can read it!) and your best bet might well be Sinclair's much abused but perfectly serviceable **ZX Printer** – if you can get it (production has now ceased, though paper will still be produced). Otherwise, buy the **Dean Alphacom**. It costs £59.95, produces a nice blue printout, and uses

rather cheaper paper. Neither of these printers needs a special interface. On the other hand, if you want something a bit more impressive then you'll almost certainly need to think in terms of a *full-width* printer – something that can manage to print 80 letters to a line rather than the 32-letter maximum for the ZX and the Dean.

Full-width printers

There are four main types:

- Daisywheel printers.
- Dot matrix printers.
- Ink-jet printers.
- Graphic plotters.

Daisywheel printers take their rather poetic name from the print element, a circle of moulded letters mounted on radiating spokes like the petals of a daisy. The wheel turns at high speed to place the required letter at the top, over the paper, and a hammer strikes it against the ribbon to print the character. Daisywheels produce beautiful printouts which can match the best electronic typewriters, but they tend to be slow (12–20 characters per second) and noisy. Some can underline, or produce bold type, but unless you actually change the daisywheel (which is possible, of course) you can't have different typefaces on the same page.

Dot matrix printers have a very different sort of print head – just a row of seven, nine, or sometimes even more tiny wires. As the head moves along the paper, one dot-width at a time, some of these wires are pressed against the ribbon (see Fig. 26.1) and a letter is formed out of the dot pattern they make, in rather the same way a letter-shape is built up on your TV screen. 'Thermal' printers work by heating the wires to create dots on heat-sensitive paper. Dot matrix printers of any kind are cheaper, faster (typically 80 cps) and more versatile than daisywheels. As well as ordinary text they can produce underlining, bold type, and even italic, enlarged, and condensed faces, not to mention superscripts, subscripts, and even,

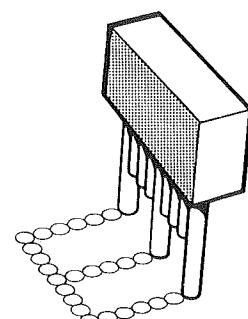


Fig. 26.1. How a letter is formed on a dot matrix printer.

in some cases, your very own character sets (fancy brushing up on your Hebrew?). In graphics mode, they can also copy a screen display, which is how we got started on this topic in the first place. Some daisywheels can, too, but not well!

The only problem with dot matrix printers is those Reader's Digest personalised letter dots, which are a dead giveaway that you've been using a word processor to type your party invitations, your job application, your homework, or whatever. Some people still frown on this (they seem to frown on anything that helps to make life a bit simpler), though admittedly the cheaper printers give them reason to by truncating the descenders (the 'tails') on letters like y and p. This is especially likely if the printer only uses a small matrix like 5×7. Still, even a 9×9 matrix gives much better results, and some of the latest machines use matrices up to 12×18, and a character set that's almost up to daisywheel standards. Some will even let you print in colour, using a special multi-coloured ribbon (guess how much *that'll* set you back!) but it's a clumsy process – and there's a better way.

Ink-jet printers work, as the name suggests, by squirting little dots of powdered ink from four tiny jets, building up a full colour image from the four primary printing colours – black, cyan (blue), magenta and yellow. The results are very attractive, though on most types of paper the ink spreads a little, so the image tends to look a bit fuzzy. The price, however, is not so nice – around £500 plus VAT. Maybe the ZX isn't so bad after all...

Ink-jet printers are really a spinoff from dot matrix printers – they use the same dots of ink but apply it in a different way. The *graphics plotter* actually draws out each letter and graphic by 'writing' it with a pen, which is lifted up and then put down again for the next letter. And colour? It's simple – you just use four pens! Plotters are at their best with graphs and charts; if you're planning to copy the ceiling of the Sistine Chapel, they're not really going to be up to the job, and text looks a bit spidery. Good ones are expensive (both the hardware and the software are fairly complex) and most of them are painfully slow.

Something else that varies a lot from printer to printer is the mechanism used to feed through the paper. There are three basic types, and some printers cater for all three! *Tractor feed* uses pins that engage in holes punched along the edge of a continuous perforated strip of paper and 'pull' it into the printer. *Friction feed* is what's used in most common or garden typewriters – the paper is gripped between rubber rollers. *Roll feed* allows you to place a full roll of paper inside the printer itself; most friction feed printers can handle this as an option.

So which one do you choose? With printers, as with everything else, there are arguments for and against every choice, so to arm you against persistent smooth-talking salesmen we've listed them in Table 26.1. These are the things you need to know to make your choice; the rest is up to you.

Spectrum calling printer, are you receiving me?

Unfortunately, the only printers that plug straight into the Spectrum are the ZX and the Dean Alphacom. For anything else you need an interface. Remember that with a more elaborate printer you've effectively bought a

Table 26.1. The pros and cons of printers.

| Type | Pros | Cons |
|-------------|------------------------------------|--|
| Ultra-cheap | Very cheap. | Illegible, slow, few facilities. |
| Dot matrix | Affordable, fast, good facilities. | Not letter quality. Can be noisy. |
| Daisywheel | Letter quality. | Noisy (often very!). Usually expensive. Slow and inflexible. |
| Ink-jet | Good colour. | Very expensive. |
| Plotters | Quiet. Line graphics. | Fairly expensive. Poor text quality. |

computer that just happens to be a printer as well – it has its own ideas of what number codes from the Spectrum mean, and you need a translator in the middle to avoid confusion. As we saw in Chapter 19, the interface is the combination of hardware and software that does the job. There are only two fundamentally different types of interface:

- (a) *serial*, which sends data one bit at a time down one wire;
- (b) *parallel*, which sends data one byte at a time, with the eight bits going simultaneously (in parallel) down eight wires.

Only two of the various standard interfaces are worth considering for use with the Spectrum. The *RS232* (or *RS232C*) is the commonest serial interface, while the *Centronics* (named after the printer manufacturer who developed it) is the commonest parallel interface. Most printers can be fitted with either type, though you often have to pay more (heaven alone knows why) for an *RS232*.

Making the choice

In practice it doesn't make much difference which interface you choose, though an *RS232* might allow you to link up a modem as well (see Chapter 27). Only the computer and the printer can tell the difference. If you have *Interface 1*, with its built-in *RS232* apparently disguised as a joystick socket, you might be tempted to use that. But be warned – Sinclair BASIC can only handle this socket rather clumsily, and you can't just use *LPRINT*. Besides, you might want to connect it to something else.

The alternative, whether or not you have *Interface 1*, is an add-on interface from another supplier. Most of these are *Centronics*, though a couple offer both *Centronics* and *RS232* in a single unit. This could be handy if you didn't want to be tied to one or the other – or if you wanted to run two printers! All these interfaces arrive with software either on board or on cassette so that you can set them up to translate *LPRINT*, *LLIST* and *COPY* into language that any printer of reasonable intelligence will understand and act upon. The code for 'pound' (£) from the Spectrum, for instance, produces an open-quote on my

printer, and there's no easy way of printing a pound sign. But with the *Hilderbay* interface, and most others with cassette-based software, you can *redefine* the characters so that, for instance, '£' really does print a pound sign.

If you want to use a full size printer you will almost certainly be using a word processing program – and by far the best for the Spectrum is **Tasword Two**, which we used to write this section. Almost all the available interfaces come with instructions on how to use them with Tasword. If yours doesn't, be suspicious. It's important.

You will also want to copy the screen display. If you're just printing up text, this is easy, but a high resolution copy of your favourite screen graphic needs a machine code routine if you don't plan to spend all night waiting for it, and it must be designed to work with your printer. Make sure your interface provides this facility, preferably in ROM but at least on cassette, and that it'll work with your chosen printer.

The final ingredient is a cable, which ought to be supplied with the interface. Most actually come as an integral part of it. The plug on the cable should fit any standard *Centronics* or *RS232* socket (see Figs. 26.2 and 26.3) on your printer – but do check before your buy. Worth looking at among the available interfaces are the *Morex*, and the *Euro Electronics Lprint III*.

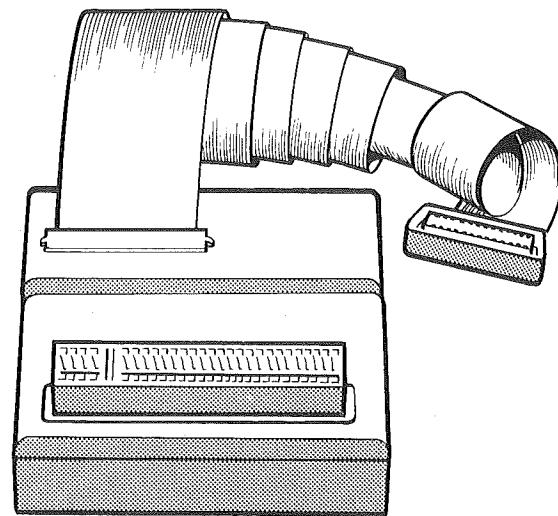


Fig. 26.2. A Centronics interface.

We can fix you up, guv...

I have heard stories about people trying to link their computer to an old teleprinter – they're obsolete, so they're cheap to get hold of. But unless you're a demon with a soldering iron you'd be well advised to forget this idea. Even if it worked, the resulting system would be bulky, noisy, slow, and thoroughly unsatisfactory. A more sensible idea might be to adapt an electronic

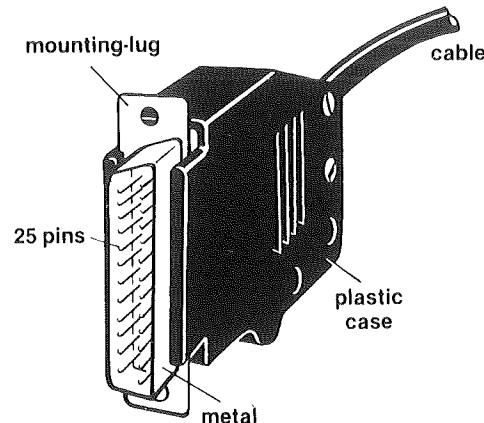


Fig. 26.3. A standard RS232 interface connector. This end connects to the printer.

typewriter, especially if you have one already. Check with your dealer that the necessary modification is possible, and check the price, too, because you could be in for a shock. The cost of printers has come down so much that it might hardly be worth it, and you can always keep the typewriter as a back-up! Until quite recently Hilderbay sold a ready modified version of the Olivetti Praxis, but this is being discontinued. All the same, they may well be able to advise you.

Some of the more recent electronic daisywheel typewriters are designed to act as printers: you simply buy the necessary interface as an extra and plug it in. If you basically want a good typewriter, which you could use from time to time as a printer, this might just be a worthwhile option – though not a cheap one. You can buy the **Silver-Reed EX43** or **EX44** for just under £300, but it will cost you at least £100 more for a parallel interface and £45 for the Spectrum interface! And after all that you'll be left with a slow (5–6 cps) unidirectional printer with no buffer, no bold, and no underline. The **Brother CE51** (£300) and **CE60** (£430) are faster and have more facilities, but the interface here – admittedly a dual parallel/serial one – costs a staggering £200. For the total money you'll be spending you could buy a very good daisywheel printer – and if you really want a typewriter then a simple ten-line BASIC program will make it behave like one by printing out everything you type on the Spectrum keyboard.

The choice is yours

You've decided what sort of printer you want. You know how to connect it to your Spectrum. You know how to get it working. But *which one do you choose?* To help you, here's a point-by-point guide to trying and buying your printer.

Try before you buy. Don't just rely on the ads and brochures – read a magazine review first, then go to a dealer and (ideally) see the printer being driven from a

Spectrum. At least use the 'self-test' facility. See if you agree or disagree with the magazine review, and check out any problems or disadvantages it mentions.

Does it have the facilities you want? If it's daisywheel, do you like the available faces? If it's dot matrix, can you get all the different typefaces you want? Can you load different character sets into it? Does it have a graphics (or bit-image) mode to let you copy pictures from screen?

Speed and noise. Remember that quoted speeds (such as 140 cps) are maximum speeds – they don't allow for things like carriage return, or the time taken for a line-feed (i.e. moving the paper up by one line). And remember that if the printer offers 'near letter quality' it's going to be a lot slower in that mode. See how long it takes to print a full page of text – and listen to the noise. You're the one who has to live with it!

Back-up and service. If the printer is made by a large and reputable manufacturer, this shouldn't be a problem. But if it's made, or imported, by a small or obscure company, make quite sure that the service arrangements are up to scratch. This is another good reason to buy from local dealers rather than mail order.

Ribbons and paper. How easily can you get new ribbons? And how much will they cost? Some printers use typewriter ribbons (cheap!). Others are very pricey – if they're carbon ribbons, ask if you can use them more than once. Paper is readily available. The commonest sort for most full-width printers with tractor feed is 8"×11" continuous (fanfold) paper. You can tear off individual sheets along the perforations, and also the strips at the sides that carry the sprocket holes for the tractor feed mechanism. A box of 2000 sheets should cost about £10–£14. Don't buy the stuff with green lines on it unless you find it irresistible (does anyone?). If the printer you're looking at will only take special paper, consider its cost, availability, and appearance.

Type style. Does it look good? Do you like it? Do you like the other options (if any)? If it's dot matrix, just how dotty is it? Are the characters well designed, with proper descenders?

Price. You know roughly how much you can afford. Is the printer you're considering really good value for money compared with other models in the same price range? Check out the magazine reviews. Can you get it more cheaply somewhere else, by mail order, for instance? You may be able to beat down the dealer's price!

Make yourself a checklist based on these points, and any others that may be important to you (strength, size, whether it'll match the new curtains...).

Chapter Twenty-seven

Calling all Spectrums...

This chapter is dedicated to putting your computer in touch with other computers – either in the same room (using the networking facility of Interface 1), or elsewhere in the country (using the telephone).

Have you heard the (true) story about the little old lady in Scotland whose phone kept ringing at one o'clock in the morning? When she answered it, all she could hear was a series of whistles and beeps. She used to hang up, because she thought it was an obscene caller. In fact it was a computer trying to reach another computer – and getting a wrong number.

If you thought that sort of thing could only happen to big mainframe computers, then you're in for a surprise. It is perfectly possible to do exactly the same thing with your Spectrum (though you're not quite so likely to get a wrong number). You can 'talk' not only to other Spectrum owners, but to home computer owners throughout the UK, to mainframe computers with massive databases all round the country, and even (theoretically, at least) to similar database computers overseas.

Why your computer should be on the phone

At first sight none of this looks terribly interesting to someone who's just bought a computer. What, after all, is the point of talking to other computers when you've only just figured out how to program the thing?

One of the first really attractive answers to that question came from an organisation called **Micronet 800**, and in their case the magic words were 'free programs'. The result was a genuine interest and a fast growing subscription list that soon made other commercial enterprises realise the value of this kind of system. Micronet 800 doesn't just offer free programs, of course – it also offers up-to-date information on the entire microcomputer scene (some of which found its way into the final draft of this book!) and a 'mailbox' facility that allows you to leave messages for fellow-users or collect messages from them. The total runs to some 40000 pages.

However, Micronet 800 is really a sort of subscription-only section of Prestel, British Telecom's own telephone data service: the jargon word for this kind of service, incidentally, is *viewdata*. Neither service is free – apart from the normal charge for using the telephone (normally billed as a local call unless you live somewhere way out in the wilds) there is a £4.30 monthly subscription for Micronet and another £1.69 a month for use of Prestel. Frankly you'll need both if

you decide to join, because most of the really interesting information the system has to offer is on the truly massive Prestel database.

Once you've got through to Prestel (of which more in a moment), you are presented with a series of menu pages rather like the menu pages used for complex programs. Each choice you make (using the ordinary number keys on the computer, with a * to indicate the start of the number and an ENTER command to finish it) brings up a more specialised menu, until you reach the specific area of information you were looking for. There's a great deal of help available from the system itself if you miskey a number or get lost, and in any case you'll have a special Prestel directory to help you find your way around.

A very large number of Prestel pages can be accessed for nothing (apart, of course, from the charges mentioned above). However, quite a few of the more specialist pages are run either by closed user groups (who charge, like Micronet, their own membership fees) or by commercial companies, who may expect you to pay up to 50p a page to see their work. You'll get plenty of warning if a page is chargeable, and if you decide to go ahead the charge will simply appear on your next telephone bill. However, if you really want to pour your money away you can also, quite often, order goods or services on Prestel; you can even run a bank account on Prestel if you want to. Normally the supplier or bank will provide a sort of viewdata form which you can fill in from your own computer with the details of what you want. If you find yourself involved with systems of this kind, however, be careful. It's all too easy to run up an enormous bill without realising what you're spending!

The least we can do is talk to each other...

If Prestel and Micronet hold no lure for you, then it's worth considering another service that only the phone can give you – data transfer between your computer and someone else's. Until quite recently this was very difficult on the Spectrum, but in the early months of 1984 the market was suddenly flooded with new devices and software to make this possible. As I said in the Introduction, this section was actually edited by phone – I sent completed chapters, in the form of machine code files, down the line to Phil Gardner in Leeds, and he was able to copy them directly onto Microdrive.

Of course, there are a good many other things you can do with this kind of communication. You can send BASIC programs, prepared messages, machine code routines, screen strings, anything the computer's internal memory is capable of storing. You can type messages to each other on the keyboard, and those messages will appear letter by letter on the receiving computer's screen display. (This may sound slow, but it's extremely useful when a crackle on the line has interrupted your program transmission.) Telephone data transmission is a lot safer and a lot faster than sending cassettes or Microdrive cartridges through the post, and if you're careful it needn't cost a lot more. So how is it all done?

Introducing the modem

The secret behind this particular communications revolution is the *modem*. A modem (short for *modulator* and *demodulator*) is really just another kind of interface, one that happens to specialise in sending and receiving messages. If the theory behind the modem holds no charms for you at all, I have some good news for you. The Prism VTX 5000 modem which I use does everything for you: once you've rung up the computer, switched on, and entered your personal code you can run happily through page after page of Prestel until you get bored. You can set up mailbox messages, send programs and data to your friends, and load up the free programs, without ever worrying about how it's all done. But if your curiosity gets the better of you, this section should help you to understand a few of the basic principles.

When you send a business letter you normally start with a formal opening (Dear Fred, Dear Sir, Sir, depending on how late his payment is) and finish with a standard close (All the best, Yours sincerely, Yours very truly. Our solicitors will be in touch with you tomorrow). For most business letters you will also expect some sort of reply: for instance if you've sent a cheque for some goods you'll expect a delivery note and a receipt with the goods when they arrive. A computer expects these things, too, and the modem's task is to make sure it gets them.

'Dear Z80A...'

In the case of your computer, the 'message' is likely to take the form of data bytes. As far as another computer is concerned, each of these data bytes is a message in its own right, so each of them needs an 'opening', a 'close', and some sort of reply.

The 'opening' is known as a *start bit*, which can, not surprisingly, be either 0 or 1. So can the 'close', or *stop bit*, at the end of every byte transmitted. Obviously, if one byte has just finished, the computer should be ready to expect the next byte! However, virtually all systems will make a 'reply' to each individual byte, by transmitting a copy of what has been received back to the computer sending the message so that its accuracy can be checked. The reply comes in on a different frequency, so the transmitting computer isn't hung up while it waits for an answer. The jargon term for all this is *full duplex*, and with their usual flair for fouling up the English language the high priests of computing have decreed that the opposite of full duplex is *half duplex*.

That's not quite all. The receiving computer also needs a way of checking that the information it's getting hasn't been corrupted – by a bad line, for instance. (Yes, even computers have trouble with telephone lines – after all, they can dial wrong numbers.) The trick here is to add an extra bit just before the stop bit (either 1 or 0) so that the decimal value of the entire code always comes out even (or odd, if the receiving system is expecting odd numbered bytes). This extra bit is known, logically enough, as a *parity bit*.

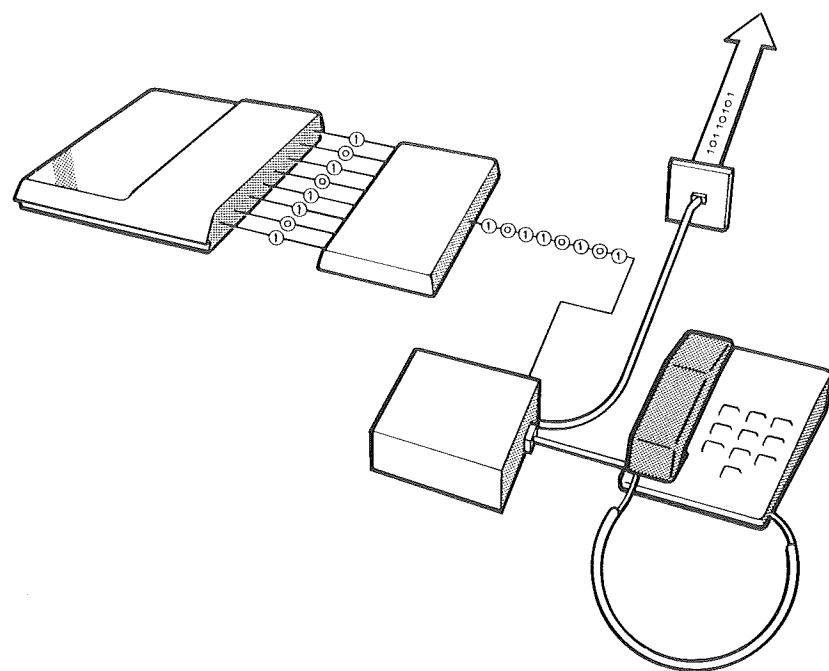


Fig. 27.1. How modems work. The parallel byte from the computer (left) is broken into serial bits by the RS232 interface. These are transmitted along the telephone line by the modem.

How it works

Obviously you can't send more than one bit of information at a time down the phone line – you need to break individual bytes of data up into their component bits before you send them. Usually this is done by an RS232 serial interface (see Fig. 27.1). Most UK systems will not accept a full 8 bits of data; they'll read the last bit as a parity bit, and if it happens to be wrong your transmission will be interrupted by an error message from the computer at the other end. In practice, this means that the computer can only send out bytes with a decimal value of 127 or less. And believe it or not, it doesn't matter.

If you're now feeling rather confused, take a quick look back at Chapter 19, where we explained how signals from your keyboard, including single-entry keywords, are read as numbers (ASCII codes) by the MPU. However, ASCII codes only cover letters, numbers and symbols, not the Spectrum's own token system for understanding keywords. That means that ASCII code numbers don't need to go any higher than 127, which is useful if you want your Spectrum to talk to an entirely different computer! As for the tokens, the numbers above 127, the interface 'decodes' them into ASCII codes. For instance, the Spectrum code for the keyword PRINT is, as we have seen, 245. If the interface picks up a 245 token it

sends out the individual ASCII codes for the letters P, R, I, N and T. Simple, really.

Once the interface has sorted out the information, it's passed to the modem itself for transmission along the phone lines. The question is, how fast? The answer depends on the precise job the modem is trying to do, and the *baud rate* (see Chapter 23) is set accordingly, but more about this later. Normally you, as operator, have to set up the baud rates in advance – one rate for transmitting data, and a second for receiving it. This isn't difficult; there are only a few standard baud rates in use, and a good modem will just have a dial or a simple software routine to cope with the selection.

Choosing your modem

There are two different types of modem currently available: the *acoustic coupler*, which looks like a sort of inverted telephone, and the *hardwire modem*, which looks like a box with knobs on.

Acoustic couplers are designed to fit over the earpieces and mouthpieces of your telephone: a speaker sends the data as audible tones through the microphone in the mouthpiece, and a microphone picks up transmitted data from the speaker in the earpiece. You simply connect the computer to the coupler and send your data directly down the telephone. Acoustic couplers have several advantages. They're light, and the portable ones are easy to carry around with you. They don't make any electrical connection to the telephone system, so there's no risk of a short circuit, which is the kind of thing that tends to make British Telecom a bit nervous. The snag is that background noise can sometimes creep through the rubber connectors and turn up as 'garbage' on the computer at the other end. And to avoid this, as far as possible, you'll need to take some trouble about setting up the connection in the first place. Then, of course, there's the simple problem of talking – you can hardly use the phone for a conversation when there's an acoustic coupler fitted to it, so you'll have to keep taking it on and off.

The so-called hardwire modem *does* make British Telecom nervous, because it's usually connected directly into the telephone system with a standard BT plug and socket. Modems of this kind must have Telecom approval before you can connect them in this way. (It's not illegal to *own* one, but this is small comfort!) The point is, of course, that the part of the modem needing electrical power must be completely isolated from the part connected to the phone system. Some suppliers cloud the issue by talking about 'approved components'. Unless the entire unit is approved, you'd better forget it. It's a pity that good old BT take so long to approve new products – they are trying to cut down the time it takes, but in a fast moving industry like this one, three months is about three months too long.

Aside from this, there are four main factors that will influence your choice of modem:

- What you want to use it for.
- What features are important to you.

- How much you are willing to pay.
- How easy it will be to connect the modem to your Spectrum.

If you only want to use a modem to swap programs and data with your friends, then there's not much point in buying something that will get you through to every viewdata system on the continent of North America. On the other hand, if you need worldwide database access then something like the Micro Myte 160 IQ/D is not going to do you much good.

Do you want the computer to do the dialling for you? It's possible – at a price – but you're going to pay through the nose for it, so make sure it's that important. Some units offer this as a plug-in extra, so you might be well advised to wait and see how useful it would really be. Some will also 'answer the phone' to another computer!

If you have an RS232 interface (for instance, one of those supplied as an option on some printer interfaces) then virtually the whole range of currently available modems is open to you – in theory. However, first you must make sure that the Spectrum can actually work with the modem you want (modems, like printers, need to speak the same language as your computer). Then you must make sure that the interface you have is suitable (sadly the one in Interface 1 isn't likely to be). Finally, you need someone to wire up the connections for you. This is not a job for the amateur electrician, as the supposedly standard RS232 connections can vary widely, so make sure the job is done for you by someone who knows his business.

Bearing all these points in mind, you can now take a look at what the market currently has to offer.

Deservedly most popular at the time of writing is the **Prism VTX 5000** modem, which uses on-board firmware to get the Spectrum in full communication with viewdata. The idea is to leave the modem connected to your computer and to the phone socket all the time – your telephone plugs into a socket in the modem itself. As soon as you power up you get a Micronet screen display, and pressing ENTER gives you a simple menu which includes the option to return to BASIC. You can also set up a message for transmission, log on or off Prestel/ Micronet, or prepare to load a program.

Controls are few and simple, just an on/off line switch and a three-position switch that lets you set the modem up for receiving or transmitting data, or for Micronet. To use the modem on Prestel, just enter your personal code number, ring up the computer, turn on the line switch, and hang up your phone (yes, really!). The Prestel logo will appear on screen, and you'll be invited to enter a pass code consisting of four numbers. Do that, and you're in – the rest is up to you.

For direct communication with another computer you need an extra piece of software, which arrives as a BASIC loader program and a machine code routine. I transferred mine to Microdrive straight away. Using this program you can send prepared messages directly to your correspondent, or transmit BASIC and machine code programs loaded from tape or Microdrive. We found from experience that any machine code routine longer than about 2K tended to be

corrupted in transmission, but there's nothing to stop you sending even quite a long file in 2K chunks.

The baud rates for the Prism can't be changed, they're fixed on one of the Prestel standards, which is 1200 for receiving information and a painfully slow 75 for sending it. This prevents you from getting into some so-called 'bulletin board' services, which use a 300/300 baud standard. Incidentally, the 300/300 standard will also give you full access to all Prestel pages. Other than that the Prism has no disadvantages, and at £100 it's definitely a sound investment. All the same, it does have competition.

When you're looking at other modems, check if the baud rates they offer will get you into the services you're interested in (the MicroMyte, for instance, is totally non-standard and will only communicate with other MicroMytes). As noted above, Prestel operates on 1200/75, and many of the bulletin boards use 300/300. Does the price include an RS232 interface? (It rarely does, so you must allow another £50 on top of the price of the modem itself.) Does the modem arrive ready-built? (There is a very cheap modem from Maplin, but it's a kit!) The **Pace Grapevine** and **Minor Miracles WS2000** modems may well be worth a look when they receive BT approval, meanwhile the **Dacom Buzzbox** is cheap and the **Tandata TM100** has some useful extras.

Putting your Spectrum on TV

If Prestel doesn't appeal, there is another source of free programs on tap, almost all day and every day. If you can call up the BBC's Ceefax pages on your television, then you may already know about 'Telesoftware'. The **TTX2000**, another product from the award-winning Cumbrian team at O.E. Ltd, can actually pick this software up and load it into your computer. Its provisional £125 asking price compares favourably with the £225 asking price for Acorn's adaptor for the BBC micro. At the time of going to press full details were not available, and the software was not ready. The production model should have this on board, and like the VTX 5000 this unit will go straight into action when you power up the system (the jargon term is *automatic boot*). It will include four channel tuners, and will provide a full colour display and, again like the VTX 5000, a 40-column display.

Net service

You only really need a modem if the computer you want to talk to is at the other end of a telephone. If it's in the same room, then Interface 1 can provide an even better link, with a speed of operation that will leave you breathless. If you have two Spectrums, two Interface 1s, and you also have two TVs (not all that uncommon these days) connect them up and start by trying out the 'net game' listed in the manual. If you have Microdrive, too, you'll find this game on the demonstration cartridge. The first time I LOADED it and transferred it to the other computer on the network everything happened so quickly that I missed it completely – suddenly there it was on the other screen!

As with Microdrive, there are three main methods of swapping information on the network:

- With standard LOAD and SAVE commands (e.g. LOAD*“n”;2) – program names aren’t used on the network
- By using INPUT and PRINT
- By using INKEY\$
- By using the MOVE command

The last three methods deserve a closer examination.

INPUT and PRINT

If you take another look at the Microdrive commands in Chapter 23 you’ll find that most of them, with minor modifications, will operate between two networked computers. The difference is that you must program *both* computers. The simplest way to describe it is to start with an example. This one uses the ‘rooms’ Microdrive file set up in Chapter 23, and incidentally demonstrates just how useful the new Microdrive commands can be:

```

110 FORMAT "n";1
120 OPEN #6;"m";1;"rooms"
130 OPEN #7;"n";2
140 FOR j=1 TO 4
150 INPUT #6;r$
160 PRINT #7;r$
170 NEXT j
180 CLOSE #6: CLOSE #7

```

As you can see, the point of this program is to read the Microdrive file and copy its contents to another computer. However, there are a few familiar commands operating in an unfamiliar way. Line 110, for instance, uses the FORMAT command to give the computer a ‘station number’. This is necessary so that other computers (there could be as many as 63!) can send messages and data specifically to you. In fact you don’t really need to FORMAT station 1 – any Spectrum fitted with Interface 1 will automatically assume that it’s station 1 until you tell it something different. Just the same, FORMATting on the net is a good habit to get into. Incidentally, it doesn’t wipe the computer, or affect the resident program in any way.

Line 120 opens the Microdrive file for reading in the usual way, but this time line 130 creates a stream to a network channel. In this case the channel is the computer with station number 2, and another buffer is set up to store this information. The FOR ... NEXT loop loads the room descriptions one by one into the computer, and line 160 PRINTS them into the network buffer. When both streams are CLOSEd in line 180, the network buffer is ready to be loaded into the computer with station number 2.

Of course, that’s only half the story. Station 2 needs some programming, too, or

you will just be left with an ugly black border around your screen display and the feeling that nothing is ever going to happen again. In this case the program for station number 2 might look like this:

```

10 FORMAT "n";2
20 DIM r$(4,20)
30 OPEN #4;"n";1
40 FOR j=1 TO 4
50 INPUT #4;r$(j)
60 PRINT r$(j)
70 NEXT j
80 CLOSE #4

```

Notice the ‘mirror-image’ effect here. In line 10 the computer is FORMATted as station 2. Line 20 sets the DIMensions of the array, much as we did in the earlier Microdrive program in Chapter 23. Line 30 OPENS stream 4 to the network channel from computer 1, and sets up a buffer to deal with the information coming in. The FOR...NEXT loop reads the data into the r\$ array and PRINTs it on screen to give you visual confirmation. Finally, line 80 CLOSEs the stream and frees the memory space occupied by the buffer. Incidentally, it also frees station 1, whose operator is probably getting a bit bored with the black border round his screen by now. One snag of network operations in general is that when two Spectrums are linked on it, only one at a time can actually do anything useful.

INKEY\$

Confusingly enough, INKEY\$ can be used to get hold of the first character in a buffer that may have nothing at all to do with the keyboard. For instance, if you program computer 2 with

```

10 OPEN #12;"n";1
20 PRINT INKEY$#12;
30 CLOSE #12: GO TO 10

```

and computer 1 with

```

10 OPEN #11;"n";2
20 IF INKEY$<>" " THEN GO TO 2
0
30 PAUSE 10
40 PRINT #11;INKEY$;#2;INKEY$;
50 CLOSE #11: GO TO 10

```

then computer 1’s operator can have the decidedly odd experience of typing keyboard input onto the screen display of computer 2 and onto his own at the same time. Notice once again that computer 2 is effectively ‘locked up’ in the loop this program creates. Incidentally, don’t try entering spaces on computer 1 or you’ll BREAK into the program! If you want to space your words, use asterisks! You can’t use the DELETE or cursor keys, either.

When all this ceases to fascinate you, try something more useful. Alter line 30 of computer 2's program to

```
30 GO TO 20
```

and NEW computer 1. Now enter this program into computer 1:

```
10 OPEN #11;"n";2
20 PRINT #11;"This is a message
   for computer 2"
30 CLOSE #11
```

RUN it, and then RUN the program in computer 2. Result? Your message will appear in full on the other screen, followed by the message 'End of file'. Of course, there's nothing to stop you sending entire screenfuls of information this way – especially useful for micro users in education.

MOVE

As noted in Chapter 23, the MOVE command can only be used with certain types of file. However, if you still have the Microdrive file 'rooms' handy, you might like to try it out on the network. Program computer 1 with

```
10 OPEN #7;"n";2
20 MOVE "m";1;"rooms" TO #7
30 CLOSE #7
```

Now program computer 2 with

```
10 FORMAT "n";2
20 OPEN #7;"n";1
30 MOVE "n";1 TO #2
40 CLOSE #7
```

Now RUN both programs – and the content of the 'rooms' file will appear almost instantly on computer 2's screen display. You could equally well have transferred the file to computer 2's resident Microdrive, or to its ZX or ZX compatible printer. Now try altering the program in computer 2 as follows:

```
30 DIM r$(4,20)
40 FOR j=1 TO 4
50 INPUT #7;r$(j): PRINT r$(j)
60 NEXT j
70 CLOSE #7
```

RUN both programs again. This time, as well as getting a screen display, computer 2 has the data file in the string array r\$. Try typing in PRINT r\$(1) and you'll see this for yourself. It's just one more example of how powerful this command can be.

Using the net

There's only been space and time here to explore the possibilities of linking two computers. But you *can* link up to 64! (Mind you, don't link the first and last into a loop, or Sinclair Research have hinted that dreadful things will happen.) I've seen at least one published adventure game for such a set-up. Are there no limits to the net?

Chapter Twenty-eight

TVs and Monitors

by *Mike Scott Rohan*

The Spectrum may have been designed to work with ordinary TVs, but its display can be pretty hard on the eyes after an hour or so. It's nice that you don't have to pay extra to get full-colour graphics from your Spectrum, but take a close look at that picture. The individual dots that make up the display seem to be constantly changing colour at the edges. This, appropriately enough, is known as *dot crawl*. Even the finest tuning doesn't help. If it's not the neighbour's motorbike, what's the problem?

Round and round the circuit board...

Most TV sets can only receive signals through the aerial socket – and the only signals they're equipped to deal with are broadcast signals from a TV station. These signals are *modulated* – the TV circuitry has to *demodulate* them to sort out the signals it needs from the 'carrier wave' that allowed them to reach the TV in the first place.

Because it uses the aerial socket, the Spectrum has to do some pretty strange things to its video signal in order to send it to the TV – in fact, it has to *modulate* the signal which is then demodulated by the TV! The unmodulated signal inside the Spectrum is much closer to what the tube electronics can cope with – and like good French novels, it loses a lot in the translation. It works, but it's not efficient. You can get an idea of the difference by comparing the Teletext picture produced by a properly equipped Teletext TV set with the broadcast version of the same picture often shown on the air outside normal programme hours. The Oracle or Ceefax pictures generated in the set itself always look clearer, steadier, and stronger.

The rather watery TV display is entirely a result of the roundabout route the signal has to take to get there. Every stage adds its own package of interference. A *monitor* is designed to take a short cut. Because it has special inputs and circuitry, it can take the signal out of the computer at a much earlier stage, and lead it directly to the video circuits. At one stage it can be taken out as a single signal containing all the colours in the image, but not yet modulated for broadcast – a *composite* signal – and at an earlier stage as three separate signals containing the red, green and blue elements of the pictures (an RGB signal). A few home computers, such as the BBC, offer all three forms (broadcast, composite, and RGB) – but the Spectrum only seems to offer broadcast. Does that mean you can't

use a monitor at all? The answer is 'no' – it's perfectly possible, using the existing circuitry, but it just hasn't been done.

Monitor output the hard way

The Spectrum uses a fairly standard design of microchip to produce its video signals, and this has extra terminals to produce a composite signal. It seems the Spectrum's original designers didn't think anyone would want monitor-quality output, and so – keeping costs down, again – they didn't provide a proper output socket. However, the signals can easily be taken out via the user port at the back, as long as you don't mind sacrificing your guarantee, and as long as you know someone who's got the necessary experience of computer electronics. This kind of work isn't easy, especially when it involves soldering, and one small slip could leave you with about a crumpled fiver's worth of junk components. If you want to risk that, fine – but please don't complain to us.

On issue 1 and 2 machines the VID strap will have to be connected to the expansion interface. The connections are detailed in the IN and OUT section of the manual as VIDEO and 0 VOLTS respectively. By the time the Issue 3 model was prepared, monitor prices had fallen, and the demand was recognised – so the connection has been made. With a proper lead this output should drive a suitable monitor perfectly well. However, if it's used at the same time as the TV output, the signal to both will be noticeably weakened. It has been suggested that taking a signal from the 0 VOLTS and Y lines can manage this without degrading the signal, but we haven't actually seen it done.

Monitor output the expensive way

Anything that can reasonably be called a monitor relies on this process of taking signals directly to the picture tube electronics. But they aren't all the same. Some rely on this alone, including most of the 'TV monitors' – which are ordinary TV sets fitted with extra circuitry and inputs to take composite or RGB signals. Many were really designed for video work – they won't give you better resolution than your TV, but they will give you sound facilities and a steadier, smoother picture. Truly 'dedicated' monitors don't usually have sound facilities, but they have a whole range of extra facilities to get more detail out of the signal. The snag is that they can't be used with videos or as TV tuners, which makes them an expensive luxury. Even the cheapest monochrome monitors cost more than a small black and white TV – more if they have green or amber screens, which are kinder to the eyes. At least you don't need a licence for them! These monitors suit pure text pretty well, but *Valhalla* or *Manic Miner* are going to look rather odd in shades of glowing green or amber!

Fortunately, they don't have to be. For £285 – which is admittedly a lot of anyone's money – you can buy the **Microvitec Cub** monitor with Spectrum interface. On the Cub, the Spectrum picture is steady, clear, and pin-sharp – you can just about count the individual pixels! The only problem we experienced was

heavy static build-up that sometimes crashed the program in the computer. The Cub gets its signals, logically enough, from the user port, and its interface there has a full through bus that is no hindrance whatsoever to connecting other peripherals. The only put-off is the price, and the minor nuisance that it can't handle the BRIGHT command. Like many professional monitors, it has no sound facility, so you can't use it with units that need TV sound such as the Currah.

Monitor output the patient way

Just as this book was going to press, **Miracle Systems** announced a new RGB interface for the Spectrum. Like the Cub's, it connects direct to the user port. Unlike the Cub's, it doesn't use any of the video signals generated by the computer. Instead, it carries its own 8K RAM, which holds a copy of the Spectrum's video display file and uses this as its picture source. The interface is expected to drive both TTL and linear RGB monitors, and would even let you have a TV picture at the same time without any noticeable loss of signal strength. On its original release this interface cost £74.75. Now it's expected to cost about £50. The unit will also include a loudspeaker and volume control to amplify BEEPs, and a MIC socket that will allow you, as the Fuller Box does, to keep the MIC lead permanently connected. Used with a linear monitor you'll even be able to get that BRIGHT command!

However, if you are patient a little longer than that – and willing to take a gamble – you might do even better. TV manufacturers seem (at long last!) to be recognising a trend in the computer market, and it may soon be possible to get TV/monitors that are almost as good as the Club – and a lot cheaper.

And finally...

... a quiet word of warning, not from me, but from a friend in the medical electronics field. Every cathode-ray tube, in a TV, a monitor, or anything else, generates X-rays. On a monochrome tube it's insignificant, but a colour signal is produced by three guns (or one very powerful one) and generates just a little bit more. Normally this wouldn't hurt a flea, and anyway you don't spend all day every day hunched in front of a TV screen. But some people – like computer book authors – do, and over a period of time even the weakest X-ray dosage can become a health hazard.

Chapter Twenty-nine

The Spectrum in Control

Where do I go from here?

If we now seem to have exhausted all the possibilities for adding new devices and new functions to the basic Spectrum, then it's only fair to warn you that everything you've read about so far constitutes the tip of an extremely large and rapidly expanding iceberg. The breadth and range of peripherals for the Spectrum is growing too fast for any one writer – or even any three writers – to keep up with. Not all the enquiries we sent received answers. Some companies had obviously folded. Some had moved. But some, as I discovered at one of the regular ZX Microfairs, were sleeping rather than dead. And one of the most fascinating products to be revived from the grave is certainly the Basicare system mentioned very briefly in Chapter 22. Then I said that it offered memory expansion up to four kilobytes (4000K), but only for advanced users. The fact is that Basicare's system is far more than a memory expansion device. It's a toolkit for tailoring your Spectrum to do almost any job that can be handled by a microprocessor. And that means *control*.

The Spectrum in charge

The subject of 'computers in control' is very large – large enough for a TV series – so I can't hope to cover it in one short chapter. On the other hand, most of the necessary theory was covered in Chapter 19; and we've already seen, especially in Chapter 24, how the Spectrum's OUT command can be used to control other microchip circuits and produce, in that particular case, controlled sounds. Looking at joystick interfaces, we've studied the IN command, and seen how the Spectrum can read in codes generated by other equipment and use them in its programs. In Chapter 25 we looked at the RD Digital Tracer and saw how it acted as an analogue to digital converter, translating voltages generated by potentiometers into numbers between 0 and 255. It's only a short step from these ideas to a fully fledged A/D converter that converts signals, or voltage fluctuations, or movements, or any other external source of information, into numbers – and uses numbers generated by the Spectrum to throw switches, move motors, or a thousand and one other useful possibilities. This isn't just pie in the sky for the average Spectrum owner; once you've grasped the basic principles, it's simple. And the hardware you need for it already exists; I've used it.

Making it happen

Let's start with a simple 'for instance'. Suppose you were going out and you wanted the lights to come on in your sitting room when it got dark. You'd check the time of sunset in your diary and find, say, that it was 7.30 p.m. Then you might use a special time switch, set to 19.30, which would switch on at the appropriate moment. You could actually do that job quite simply on the Spectrum.

Start by connecting an analogue-to-digital converter and a switching unit to the user port. I used the ADC 8.2 and REL 4.2 units from Harley Systems. Next, connect a light-sensitive cell to the converter. Using the instructions supplied with your converter you can quite easily get a screen display showing the readings produced by the cell. Find out the readings given in daylight and when the room is dark (draw the curtains!). Now connect up the light to the switch unit – this is quite simple, but do make sure that the unit is designed for mains voltages! Now all you need to do is write a simple loop program that checks the reading coming in from the light meter using an IN command. If the reading drops below a certain level, then the program moves on to an OUT command that operates the switch and turns on the light. And there you are. You can keep the program ready for use anytime, and forget about looking up sunset times in your diary. It's practical, simple, and effective – after all, you don't use your computer when you're out! Incidentally, Fig. 29.1 shows a graph display in response to input from a light-sensitive cell.

Suppose you go one stage further? There is space on the REL 4.2 to connect four different mains devices, and space on the ADC 8.2 for eight 'sensors' – anything from heat measuring devices to movement detectors. With one fairly

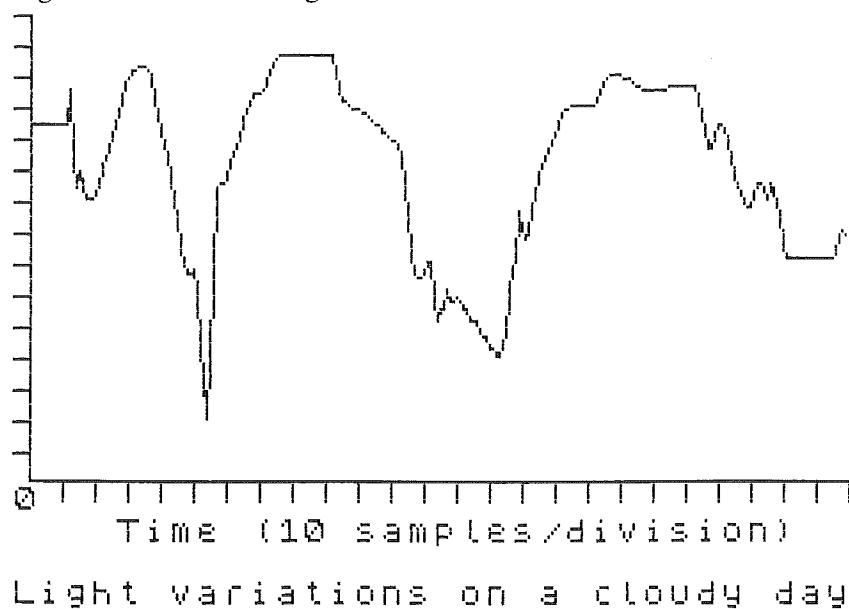


Fig. 29.1. A graph display in response to input from a light-sensitive cell connected to the Blackboard Electronics A/D converter.

simple program you could use the Spectrum not only to control the lights, but also to set off an alarm if it detects movement in the house, or if a heat sensor detects a fire.

In much the same way, your Spectrum could control a robot. Most robots in production today (and there are plenty) use something called a *stepper motor*; its movements can be precisely controlled, one 'step' at a time, by applying controlled bursts of power. For instance, if your analogue-to-digital converter was connected to output port 131 you could arrange for OUT 131,90 to turn the motor through 90 degrees – and your robot would duly turn 90 degrees in response. Minus numbers could be used to reverse direction: so OUT 131, -90, would turn him back to where he was before. This isn't fantasy: stepper motors can be bought, and you can always steal some of the kids' Fischer Technik to build the robot! If the kids want to play too, how about writing a program to control a model railway? A switch unit like the REL 4.2 could easily perform simple jobs like switching points and signals.

The available hardware doesn't stop there. I also tested another analogue-to-digital converter from Blackboard Electronics (distributed by Philip Harris) which included a microphone and a full ECG kit (as the deadline for this book drew ever nearer I was able to monitor my incipient heart attack). The demo tape included a 'fun' program that involved shouting into the microphone in order to keep a helicopter on screen 'airborne' – but it wouldn't take an ace programmer to link, for instance, the microphone to a piece of software that gave an insistent flashing warning whenever the kids's stereo was making too much noise, or even (using a switch unit) turned it off! (Cruel but effective!) It's also worth remembering that the William Stuart Big Ears unit is really an analogue-to-digital converter – the software simply sets up patterns of numbers corresponding to the sounds of a word in memory, and when the unit gets a reasonable match for that word, it 'recognises' it.

Glanmire Electronics supply a time controller that can be plugged into the user port. You can use it two ways. By taking readings from it and checking them in the program, your Spectrum can become a time switch. By running a different program and incorporating readings from the time controller, you can get a precise record of when something happened.

All these ideas may sound mildly useful on their own – but imagine combining them! That's possible, too, especially if you use EPROMS. EPROM stands for Erasable Programmable Read Only Memory, and it's a very special kind of microchip that can be programmed with your own software routines and then switched in to replace part or all of the Spectrum ROM – rather like the cartridge ROMs for Interface 2. The difference is that you reach it with an OUT command. You can have a whole nest of EPROMs connected to the Spectrum and access them one at a time – a whole program library waiting to be cut in. My sample board from Orme Electronics included a renumbering routine, another that could delete a selected block of program lines (very handy) and another that would scroll graphics from side to side. EPROM programming is a job for the knowledgeable, and certainly not something that I would tackle tomorrow – but it's possible, and it

opens up a whole new area of control for the Spectrum. The real beauty of an EPROM is that it doesn't use any of the Spectrum's own memory space, so you can have a fairly complicated program in memory and call up elaborate subroutines on EPROM that you'd never be able to fit into the computer otherwise.

Building the tower of power

I started this chapter with the Basicare system, and now that we've seen what's possible in this area it's time to look at it again. The foundation of the Basicare system is the so-called Persona unit – and to fit it to a 48K Spectrum you start by effectively cutting out the 32K of memory you bought the thing for! On its own the Persona does very little, so you may wonder why you should perform this act of sacrilege, but in fact the unit is a buffered (i.e. protected) interface for all the other modules in the Basicare range. Once it's fitted, you can start stacking... and stacking... and stacking. (That theoretical 4000K would form quite a formidable tower.) Mind you, after five units you're going to need some more power from somewhere – the poor little Spectrum mains unit can only manage just so much!

So what, you may ask, are the units? The short answer is almost everything I've discussed above, including a 64K RAM pack, an 8-channel analogue-to-digital converter, a sound synthesiser, a clock, an 8K ROM unit that can replace part of the Spectrum ROM when required to do so, a 24-line input/output board, a Centronics interface, a Minimap module that lets you select which chunk of all this you need to work with at any given time, and so on and so on. If you seriously want to turn your computer into the heart of a control and communications system, then this is one very well structured way to do it. Believe it or not, the system was originally designed for the ZX81 – on a big Basicare system it must have looked like a swallow's nest on the side of a skyscraper. The Basicare system is not a toy, and it's not for dabblers – you need to know what you are doing. But it's odd how computers bought as toys by dabblers become serious tools for serious users.

The Spectrum as chameleon

Throughout this section we have seen the Spectrum, once almost despised as a toy, turned to a surprising and remarkable range of practical, not-so-practical, and downright impractical purposes. My Spectrum, in the course of writing these chapters, has been a word processor, a data processor, a graphics generator, a database access keyboard (to Prestel), an electronic mailbox, a games machine, a domestic controller, a sweet talker and music maker, a record-keeper, and a constant source of surprise and enjoyment. It has been a test-bed for new ideas and new equipment, a source of relaxation and a superbly flexible creative tool. I've learned new techniques and developed new ideas. Some of them – most of them – I have tried to pass on in these chapters.

The Spectrum isn't just a small black box full of circuitry. It's an endless source

of interest, new ideas, and new forms of enjoyment. If some of the things you've been reading about have caught your attention, go out and try them for yourself.

Good luck – and good computing!

Part 6

Introducing Machine Code

by Ian Sinclair

Introducing Machine Code

Many computer users are content to program in BASIC for all of their computing lives. A large number of others are eager to find out more about computing and their computer than the use of BASIC can provide them. Few, however, seem to make much progress to the use of machine code, and I think this is because so many books which deal with machine code seem to assume that the reader is already familiar with the ideas and jargon words of machine code. Also, these books tend to treat machine code as a study in itself, leaving the reader with little clue to the application of machine code to his or her own computer.

This section has two main aims. One is to introduce the Spectrum owner to some of the details of how the Spectrum works, so allowing for more effective programming even without delving into machine code. The second aim is to introduce the speed and power of machine code by means of simple examples. I must emphasise the word 'introduce'. No single book can tell all about machine code, and all I can claim is to give you, the reader, enough information to get started. Getting started means being able to write short machine code routines, understand such routines printed in magazines, and generally make more effective use of your Spectrum. It also means that you will be able to make effective use of books on machine code programming – these are the books which are your entry to much more advanced work. From there, complete mastery of machine code programming is just a short step.

Together, understanding the operating system of the computer and having the ability to work in machine code can open up an entirely new world of computing to you. Understanding the operating system allows you to do things like renumbering program lines, changing PRINT instructions to LPRINT with one command, altering the key-press BEEP, or printing out a list of all variables. Writing machine code allows you also to take complete control over the computer system so that you can carry out tasks like reading ZX81 tapes, driving serial printers from the cassette port, speeding up actions like denary/hex conversions or screen graphics. I must emphasise that this section does not consist of programs – it consists of explanations, because it is only by 'doing your own thing' that you will ever learn effective machine code programming.

No workman can operate without good tools. The tools that I have used are the Spectrum itself, a Trophy CR100 cassette recorder, a Philips 14CT3005 television receiver and the ZX printer. The software tools were the 'dpas' Disassembler from Campbell Software, which allowed me to investigate the Spectrum operating system, and the ULTRAVIOLET Assembler from ACS Software, which allows a

much faster and easier coding of programs written in the intermediate assembly language.

Many of the underlying concepts you need to understand in this section have already been introduced in earlier parts of the book, particularly Chapters 19 and 27 – so if you skipped over Part 5 you might like to go back and take a look at those chapters before you start. However, if you've been working your way steadily through the book the moment of revelation is at hand! The chapters that follow should (at last!) help you to understand how and why the Spectrum works as it does, and provide a firm foundation for more advanced work.

Chapter Thirty

Inside the Case

Don't take the title too literally, you don't need to open the case! What I mean is that in this chapter we are going to look at how the Spectrum is organised to load and run BASIC programs, and we shall in the course of this discover the meanings of some of the numbers and the cryptic titles that occur in Chapter 25 of the Spectrum manual.

Let's start with a simplified version of the action of the whole system – simplified in the sense of omitting a lot of detail that would just be confusing at this stage. The ROM of your Spectrum consists of a large number of short programs – *subroutines* – which are written in machine code. There will be at least one of these machine code subroutines for each keyword of BASIC, and some of the keywords may require the use of many subroutines. When you switch on the Spectrum at first, the piece of machine code program that is carried out is called the 'initialisation' section. This is a long piece of program, but because machine code is fast, carrying out instructions at the rate of several hundreds of thousands per second, you see very little of it – the only evidence on the screen is the black rectangular pattern that appears just before the Sinclair Research copyright notice. In this brief time, however, the size of the RAM has been checked (in case you added some extra chips last night). It has been cleared of any unwanted bytes, a process that is necessary because of the effects of switching memory off and then on again. When a RAM memory is switched off, all the units of memory revert to the \emptyset setting, as you might expect. When you switch on again, however, there is no guarantee that they will all stay that way. In fact, roughly half of them go to the '1' setting, completely at random, so that if you were to read each byte of memory at the instant when the computer was switched on, you would find that each byte of memory stored a number between \emptyset and 255, quite at random, with no rhyme or reason to the numbers. This kind of thing is called *garbage*, and one of the tasks of the initialisation program is to replace each of these garbage bytes by a \emptyset , by deliberately writing a \emptyset into each unit of the RAM memory in turn. As a result, if you switch on and PEEK a RAM memory address above about 239 $\emptyset\emptyset$ you will find that the content of each byte is zero.

Initialisation consists of a lot more than this, however. Of the 16K of RAM in the smaller Spectrum, a large chunk is used by the operating system, meaning that the machine code routines use the RAM to store quantities that may have to be altered at various times as the program is used. Addresses from 16384 up to 23755 are used in this way, and that's more than 7K of memory (1K=1024 bytes) out of your 16K before you have typed a single character of BASIC. In addition, once

you start to enter a BASIC program, more RAM has to be set aside, this time at higher memory addresses for storing quantities that are needed to make your program run. Every time you 'declare a variable', for example, by using a line such as:

LET n=20 or LET a\$="Smith"

you cause several bytes of memory to be taken up by entries which store the variable name, n or a\$ or whatever you used, and the quantity or the characters (20 or Smith). You saw this in action in Part 4! The piece of memory that is used for these purposes is called the Variable List Table (VLT), and it is kept immediately above the space used by your program. Add one line to your program, and the VLT has to be shifted up to make more space. Delete a line, and the whole VLT list moves down to lower memory addresses.

This is a type of behaviour that has to be controlled rather carefully because the computer must at all times keep a note of where this piece of memory is located. This is done by an entry in one of the pieces of RAM reserved for such an 'index'. Since the principle is used very extensively, we might as well examine this particular example in detail.

The important addresses that the computer must keep a track of are stored between 23552 and 23733, and the starting address for the variable list table is stored at the address 23627 and 23628. Now the addresses that we want to store will also consist of five-figure numbers like these, and we know already that one byte of memory can hold a number which is between 0 and 255 inclusive. If you want to store numbers that are greater than 255, then you need at least one more byte, and the scheme that computers use to store address numbers is to take one more byte to store the number of complete 256's in the number that is to be stored – using a scale of 256 rather than a scale of two or ten. If we had two bytes stored which read, in order, 10,1; this would mean $10 + 256*1=266$. If the bytes were, in order, 24,32, this would represent the number $24 + 256*32=8216$. Note that the order of storing the numbers is low byte, then high byte. To find the address number that is stored in the addresses 23627 and 23628, then, we need to use the formula PEEK 23627+256*PEEK 23628. The result of this is an address – and it's the address of the starting point for the variable list table.

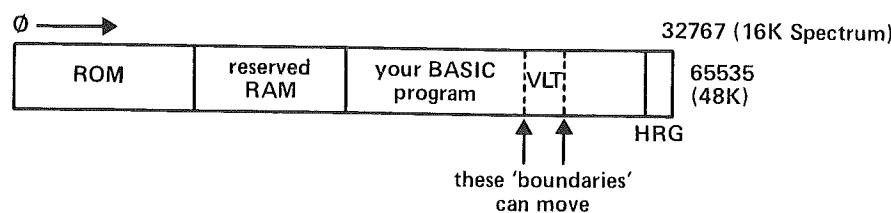


Fig. 30.1. The position of the variable list table (VLT) in the memory is not fixed – it is placed just beyond the BASIC program, and will move up or down as the BASIC lines are added to or deleted.

Try typing into your Spectrum:

100 PRINT PEEK 23627 + 256* PEEK 23628

and run this. Note the number that is printed. Now add some lines such as 10 LET n=12 and 20 LET a\$="Smith" and RUN again. You will get a larger number printed, because the start of the variable list table has been moved up to a higher memory address to make room for the additional lines of BASIC. Add still more lines of BASIC and the start of the VLT has to move even higher. Delete some lines, and the VLT can move lower again. When the memory addresses that are allocated for some purpose or other are stored in this way, we say that they are 'dynamically allocated'. That's also why the manual warns you not to alter the address of the VLT (by a POKE command), because this would cause the computer to lose track of some of its variables. Try it – use the command:

POKE 23627,203:POKE 23628,92

Now try to RUN and watch the chaos. Switch off and on again if your Spectrum hangs up and refuses to respond to keys. What you have done is to mislead the fast but brainless microprocessor that runs your Spectrum.

This, incidentally, is an introduction to the POKE command. The first number following POKE is the address whose byte you want to change, the second is the byte you want to put into that address. Later on, we'll look at the POKE process in more detail, but for the moment, note that the number which is POKEd into memory in the example above is 23755 – where a BASIC program starts rather than where the VLT should start.

Now try something very much more constructive when you have regained control. We shall now take a look at what the Variable List Table contains. As you might expect, it has to contain the 'name' of the variable and its value, but the Spectrum does some coding of values so that it can find and use the values when it needs to.

To start with, the first byte in the variable list table is always a number greater than 40 denary. The reason for this is so that the computer can detect the *last* line of a program. Program line numbers are restricted to the range of 0 to 9999 denary, and since 9999 would be coded by the two bytes 15 (low byte) and 39 (high byte), the high byte of the number can never exceed 39. Since the high byte of the line number is always the *first* byte of a line, the computer can check this byte, and if it is 40 or more, this will signal the end of the BASIC program.

This, however, means that some care is needed to ensure that the first byte of an entry in the VLT is a number greater than 40, which is easy enough, because the ASCII code for a valid variable name must be a number greater than 40. The other point is to code these variable names so that the computer can distinguish the different types of variables from each other. This is done by using only five bits of the first byte for the variable name letter code, and using the top three bits for a coding for the variable type (number, string, array, etc.). This needs some more detailed explanation.

For a simple number variable which consists of one letter only, the first byte of

its VLT entry is just the ASCII code for the name. The manual shows this in rather a confusing form – with 96 subtracted from the ASCII code and then added on again. It ensures that each entry starts with the bits $\phi 11$, which is the coding for a number variable whose name is a single letter. The value for the number is then contained in the next five bytes after the letter code. Unless you are curious or of mathematical inclination, it doesn't do to enquire too closely as to how the value of a number which is fractional or negative is coded! If we stick to integers, meaning positive whole numbers less than 65535 as far as Spectrum is concerned, then for an integer less than 256, the number is stored as a single byte, the third byte following the coded name. If the integer lies between 256 and 65535, then two bytes are used – the third byte holds the less significant part of the number and the fourth byte holds the more significant part (the number of 256's). The fact that integers still need five bytes for storage is one of the factors which makes Spectrum BASIC so much slower than the BASIC of machines which can treat integers differently.

```

10 LET a = 10
20 LET b = 11
30 LET c = 12
40 FOR n = 0 TO 30
50 PRINT PEEK (( PEEK 23627 + 256* PEEK 23628)
  + n) ;"      ";
60 NEXT n

```

Fig. 30.2. A program to investigate the storage of integers.

Fig. 30.2 shows a simple program which allows you to investigate the storage of some integers, and displays the results on the screen. If the name of the number variable consists of more than one letter, another type of coding is needed, because otherwise the computer would still read the five bytes following the first character as if they were the value. The scheme that is used in this case is to add 64 to the ASCII value of the first letter of the name, then store each subsequent character in normal ASCII code form until the last character, which has 128 added to the ASCII code. This is done so that the computer can recognise the end of the name and then count out the next five bytes as the bytes used to hold the value. Fig. 30.3 shows an example of this method that you can try for yourself.

A quick look at a variable which is not an integer is instructive, and Fig. 30.4 shows such an example. What is of interest now is not the coding of the letter

```

10 LET julie = 23
20 FOR n = 0 TO 17
30 PRINT PEEK (( PEEK 23627 + 256* PEEK 23628)
  + n) ;"      ";
40 NEXT n

```

Fig. 30.3. How a long number variable name is stored.

```

10 LET a = .5
20 LET b = .25
30 LET c = .125
40 FOR n = 0 TO 30
50 PRINT PEEK (( PEEK 23627 + 256* PEEK 23628)
  + n)
60 NEXT n

```

Fig. 30.4. Non-integer variables. The examples have been chosen to give reasonably simple results.

variable name, but the fact that all five bytes are used for coding the value. This indicates that the value which is stored is never exact, so that when you have a program that uses fractions, there will be some 'rounding' errors. These will seldom show on the screen, because the value that is shown on the screen is not of as many decimal places as the stored value (it has been rounded up), but it can cause trouble in BASIC statements which look for identical values. Try the program of Fig. 30.5 to see what I mean – the fact that the computer does not recognise the numbers as being equal is due to the small differences in the stored values which do not appear on the screen. It's like saying that 1.00000007 is not

```

10 LET n = 1/1000
20 LET p = 10/10000
30 PRINT "n = "; n; " and p = "; p: IF n = p
  THEN PRINT "Equal"
40 IF n <> p THEN PRINT "Not equal"

```

Fig. 30.5. The computer may not have the same idea about equality as you do! This is caused by 'rounding errors'.

equal to 1.00000008 – if we print only the first six places after the decimal point, then both numbers appear on the screen as 1.000000, and the computer is detecting a difference of one part in 100 million! No-one needs accuracy like this, and it causes trouble only when you have equality statements in programs. A good way of dealing with the problem is always to equate quantities which have been rounded, as for example, by statements like:

```
IF INT(10E4*a)=INT(10E4*b) THEN GOTO ...
```

which will perform the GOTO if the numbers are equal to an accuracy of four decimal places – good enough for all but the most exacting purposes.

Fig. 30.6 shows what happens when the VLT stores a negative integer – once again you don't need to be able to follow the coding exactly, but note that three bytes are used. Turning from number variables, Fig. 30.7 shows how a string variable is stored. The variable name is coded as the ASCII value minus 32, and the string sign, \$, is not included in the coding. Spectrum permits only single letter names for string variables, so the complications of extra letters do not arise in this case. The string name is followed by two bytes which store the number of

```

10 LET a = -12
20 LET b = -176
30 LET c = -255
40 FOR n = 0 TO 17
50 PRINT PEEK ((PEEK 23627 + 256*PEEK 23628)
  + n);";";
60 NEXT n

```

Fig. 30.6. How negative integers are represented in the VLT.

```

10 LET a$ = "Sinclair"
20 FOR n = 0 TO 17
30 PRINT PEEK ((PEEK 23627 + 256*PEEK 23628)
  + n);";";
40 NEXT n

```

Fig. 30.7. Storing a string variable.

characters in the string so that the computer can be instructed how many bytes to read. Since two bytes are used to store the string length (most computers use only one, which speeds up string handling), Spectrum permits very long strings at the expense of a byte which is wasted if strings of normal length are being handled. The length bytes are followed by the characters of the string using normal ASCII codes.

The coding used for arrays or numbers or characters (string arrays) is rather more complicated (see Spectrum manual pp166-168). While we are on the subject of storing variables, however, we can explain the numbers that you will have noticed on the screen following the VLT for the numbers and strings that we have been looking at. These are the numbers associated with the FOR ... NEXT loop that was used to print out the values. Fig. 30.8 shows a program that consists only of such a FOR ... NEXT loop, arranged to print out its own VLT. The variable

```

10 FOR n = 0 TO 17
20 PRINT PEEK ((PEEK 23627 + 256*PEEK 23628)
  + n);";";
30 NEXT n

```

Fig. 30.8. How the values for a loop are stored.

name that is used for the variable of the loop, in this example n, is stored as the ASCII code plus 128, and its value is stored in the next five bytes in the usual way. This value will change on each pass through the loop, so that the value which you see printed on the screen is the value that happened to exist at the time of printing. Since this value is the fourth character printed (with n starting at 0) then the value is 3 at this time, but will change to 4 after the NEXT, and so on. The next set of five bytes is reserved for the final value (18 in the example) and the third set of five bytes is for the STEP value, which will be 1 unless we have specified something else. Two bytes near the end of the block specify the number of the 'looping line',

the line number to which the loop returns if the loop has not ended, and the last byte specifies the statement number within that line, in case the FOR ... NEXT loop has started inside a multi-statement line like:

```
10 LET a=2 : LET b=3 : FOR n=1 TO 6
```

A total of 19 bytes are used; another reason for the comparatively slow running of Spectrum BASIC. Many machines permit statements like:

```
FOR N% = 1 TO 16 STEP a%
```

which specifies integers stored in two bytes each; such loops can run very much faster and take much less memory. Spectrum does not permit such options.

Program entry

We've seen now that two sections of the RAM are used in ways that are controlled by the computer. The lower area of RAM is reserved for use by the system, meaning that it will be reserved whether you type a BASIC program into Spectrum or not. The upper section of the RAM is reserved mainly for use with a BASIC program, and the less call your BASIC program makes upon it, the more memory will be left for you to program with! It's time now to take a closer look at what goes on when you enter a program.

As you know, the computer shows the line you are typing on the bottom lines of the screen, before you ENTER. You can delete parts of this line or lines, because the whole section is not put into the normal final resting place in memory, which starts at 23755. Instead, it is stored in a temporary space, a piece of memory that is called a 'buffer' (see also Chapter 23). This is another piece of dynamically allocated memory which has to shift up each time another line is added to the BASIC program. That, incidentally, is why it can take a noticeable time between pressing ENTER and seeing the 'K' cursor appear again on the bottom line.

Suppose, for example, you typed:

```
10 LET n=12
```

Until you press ENTER, this is treated as a temporary entry only, and is placed in the buffer space. If you had omitted the line number, then the line would *never* occupy any other space, but with the line number present, the computer is programmed to place this line into the main program memory space when ENTER is pressed. Where is the start of this program space stored? It doesn't usually change, but nevertheless its starting address is stored at 23635, 23636, so that we can find the address of the first byte of the program by using:

```
PRINT PEEK 23635 + 256* PEEK 23636
```

and this number is usually 23755. If you alter this address by POKEing different numbers into 23635 and 23636, then your computer will not recognise that there is a program stored, because it can't be listed or run unless the correct starting number is present. Furthermore, if the new address you have put into these two

bytes is well above the first part, you could write a second program in this other piece of memory, list it and run it. By restoring the original address bytes into 23635 and 23636, you could then list, run and work with the first program, ignoring the second. You would have to be sure, however, that each program carried its own variable list table, and that the variable list table address was correct before running a program.

Back to normal programming, though. What does this line of program look like when it is stored in the program space? We can find out by using a short piece of BASIC command, assuming that you have the line 10 still in the memory:

```
FOR n = 23755 to 23785: PRINT PEEK n;"":NEXT n
```

This is a direct command so that it will not get mixed up with the line that we want to investigate.

Fig. 30.9 shows what you can expect to see – a string of numbers, of which the key is the ASCII code for n, which is 110. We can pick this out in the listing, and

| | | | | | | | | |
|----|----|-----|----|-----|-----|----|-----|----|
| Ø | 1Ø | 12 | Ø | 241 | 11Ø | 61 | 49 | 5Ø |
| 14 | Ø | Ø | 12 | Ø | Ø | 13 | 234 | |
| Ø | Ø | 222 | 92 | Ø | Ø | Ø | 233 | 92 |
| Ø | Ø | Ø | 1 | Ø | | | | |

Fig. 30.9. The appearance of a line of BASIC in memory – this BASIC program allows you to look at how it is stored.

knowing that 110 means 'n', we can deduce that 241 is the code for LET (there is a list of the codes for keywords in the manual), and we should know that 61 is the ASCII code for the equality sign. We can also see that the number 12 is coded as two bytes of ASCII code, 49 for 1, 50 for 2. That codes LET n=12 into 5 bytes, but there are 4 bytes ahead of LET and a large number of them after, with a 13 at the end of the string. The end byte is 13, the code that is used for the ENTER key, and the bytes between the 14 and the 13 are the coded form of entry which will be used in the variable list table: 14 is used as a signal that what follows is in correct number code form, as distinct from ASCII form. The four bytes that precede the LET code are of more interest to us at the moment, however, because they illustrate yet another way in which the computer uses the RAM to keep a track of what is going on. The first two bytes are the line number. Unlike all other two-byte numbers that the computer uses, these are in conventional (to you) order, with the high byte (the number of 256's) first, and the low byte second. This is done, as we explained earlier, so as to allow the computer to sense the end of a program by looking for a byte greater than 40. Your line 10 should give rise to bytes Ø, 1Ø, but if you had started with a line 10ØØ, then the numbers would have been 232,3, because $3 * 256 + 232$ equals 10ØØ.

What of the next two bytes? If you typed the line just as I have shown it, then the next two bytes will be 12,Ø. This is the more usual low then high order of bytes, and the number twelve is the total number of bytes of code in the line, including the text (LET n=12), the VLT entry of the code for n and five bytes of value, and the

13 byte at the end of the line, but excluding the four bytes at the start. Why is this length number needed? The reason is simple, like the action of the microprocessor. If the computer keeps a count of the bytes as it goes along, then the number that is stored in its address counter after it has read these two 'length-of-line' bytes will be the address of the first useful byte (LET in our example) in the program line. When this number has added to it the number stored in these two 'length-of-line' bytes, the result is the address of the start of the next set of the four bytes that give the line number and length for the next line. This is how the computer gets from one line to the next, how the correct number of bytes can be transferred to the 'bottom line' of the screen (a different part of memory) for editing, and how lines can be re-sorted into order.

Every time you type a new line of BASIC and ENTER it, then, you use up five bytes of memory for the line number, the length number, and the ENTER (code 13) byte. This is called the line overhead, and the interesting feature about it is the use of two bytes for the line length. Most computers use only one byte here, permitting lines of no more than 255 characters, and since most lines are much shorter than 255 character, this byte is usually zero. There is a wasted byte in each line, therefore, unless you are in the habit of typing quite impossibly long lines. You can overcome some of the line overhead problems by making full use of multi-statement lines, because the line number and length bytes are used only when a new line number is started, not when you use the colon to separate statements. Try this one:

```
1Ø LET a=12 : LET g$="Sinclair"
```

and then look at the line by using:

```
FOR n=23755 TO 23785: PRINT PEEK n;"":NEXT
```

to look at the bytes that have been stored. The result is shown in Fig. 30.10. Now try POKE 23756,2Ø, and then LIST your program line. You have changed the line number to 2Ø simply by changing the line number byte! Now try some more POKEing. Type and ENTER:

```
POKE 23759,234 : POKE 2376Ø,13:POKE 23752,2 : POKE 23761,238
```

and then LIST your program. It now appears to be:

```
2Ø REM
```

There is no trace of the old program, though bytes from it are still in the memory. This illustrates very dramatically what POKE, which places bytes directly into the memory, can do. We'll look at the syntax of the POKE command later.

| | | | | | | | | |
|-----|----|-----|-----|-----|-----|-----|-----|----|
| Ø | 1Ø | 27 | Ø | 241 | 97 | 61 | 49 | 5Ø |
| 14 | Ø | Ø | 12 | Ø | Ø | 58 | 241 | |
| 97 | 36 | 61 | 34 | 83 | 1Ø5 | 11Ø | 99 | |
| 1Ø8 | 97 | 1Ø5 | 114 | 34 | 13 | | | |

Fig. 30.10. The result of storing a multi-statement line.

Running a program

When you have typed a BASIC program and entered each line into memory, the arrangement will be as we have seen, with the line number bytes, line length bytes, the keyword tokens and ASCII codes that make up the line, the VLT entry codes, and the code 13 that signals the end of the line. As each line is entered, which includes transferring it from the buffer address to its more permanent resting place, it is checked for syntax, and the usual error warnings will be delivered if such an error is detected. The ZX family of computers is almost unique in carrying out this syntax check before the line is entered, and this checking can save much tedious work later. Many computers will accept syntax errors quite happily, and will report them only when you try to RUN the program, which is rather late to find out.

Now, however, we want to look at the way in which the coded BASIC lines in the memory are treated by the computer when you press the RUN key and follow it with ENTER. Before you get to this stage, remember, the computer has stored a lot of information about your program in the reserved portions of its RAM. All of the variables, for example, will be ready to place into the VLT, whose starting address is calculated and stored ready. The start of the first line of BASIC is held in addresses 23635, 23636, and the end of the BASIC program bytes is where the VLT starts, as we have seen.

Now computers work to quite inflexible rules. The first byte of a line, after the line number and the line length bytes which are really just part of the computer's housekeeping system, is always a command word token. The Spectrum ensures that the first byte in each line is a command token by its entry system, which checks for this and will not permit a line to be entered if this is incorrect. This avoids having to check this when the program is RUN, as other computers have to do. What happens then depends on what the instruction happens to be. If it is a simple assignment, like LET n=12, then an entry is copied into the variable list table from the part of the line that starts with the code number 14.

A complex assignation, like LET y=2*n needs some more action. The code token for LET calls up a machine code subroutine so that the name y is entered into the variable list table, but this routine has to be interrupted to carry out the multiplication routine, 2*n. This uses another set of machine code subroutines which first search through the variable list table to find the name n, extract its coded value, carry out the multiplication, and then return to the task of assignation, putting the answer that has been worked out into five bytes of the variable list table that follows the variable name entry.

These two simple examples illustrate two important actions that take place during a RUN (RUN-time actions), making entries into the variable list table, and reading entries that have already been made. Most of the simple BASIC actions are of these types. For example, the command PRINT a\$ will start with the token for PRINT, which calls up a PRINT subroutine (one of the longest and most complex in the ROM). This subroutine will in turn call up another one which will search for the entry for a\$ in the variable list table and which will, once a\$ is found,

store the number of characters in a\$ and the address of the first character. Now that the machine has some clue as to where to find the ASCII codes corresponding to a\$, and how many of them are to be printed, it can complete the PRINT routine. This is quite complicated because of the design of modern computers like Spectrum. At one time, the PRINT routine only had to copy the bytes of the characters from their places in the program or VLT memory into another set of memory addresses called the video memory. This was a physically separate chunk of memory, so that a computer like the 16K TRS-80 actually had 16K available for the user – the 1K that served the screen display was separate and not counted in this total. The convention that is used nowadays is to use a lot of the RAM for what the Spectrum manual calls a 'display file' – another form of video memory. This is part of the 16K of RAM which is used to store the bytes that dictate the shapes of characters and, on the Spectrum, each character needs eight bytes of RAM. These bytes, however, are not stored consecutively in the memory. Of the eight bytes per character, each is stored 256 places beyond the previous one. You can check this for yourself by using the simple BASIC program of Fig. 30.11, which POKEs bytes into the memory locations that are used for the first character location on the screen.

```
1Ø FOR n = Ø TO 7
2Ø POKE 16384 + 256 * n, 68
3Ø NEXT n
```

Fig. 30.11. POKEing to a screen location. This requires values to be POKEd to memory positions 256 bytes apart.

The bytes which are used to make up the characters that are printed on the keyboard are stored in the ROM and, as you might expect by now, the starting address of this set of bytes, the character set, is kept in the reserved RAM. The address stored in 236Ø6, 236Ø7 is 256 less than the address of the start of this character set (so that the program which uses it can run in a loop which starts by adding 256 to the address), which is at 15616. The character set is in the ROM, but since its starting address is stored in the RAM, we could change this address to point to a new set of characters which we can store in RAM – this is what we do when we create the user-defined characters, in fact. The PRINT routine makes use of these stored character bytes, and sends copies to the correct addresses in the display file at the start of RAM, so that the TV circuits can then generate the signals which form the shape of the character on your receiver. At the same time, the 'housekeeping' routines swing into action to ensure that the next character will be either in the next space along, or placed wherever it is commanded by TAB or AT, or put into the start of the new line.

It would be possible to fill volumes by examining each action of the Spectrum in detail, but there isn't much point as far as we are concerned here, because they all follow pretty much the same general pattern. A command word is represented in the memory by a token which at the RUN time is used to call up a subroutine, which will in turn make use of the variable list table and other subroutines to carry

out its work. Sometimes a subroutine may have to 'hang up' – for example, a routine that carries out a PRINT $n*k$ can't get on with printing until the result of $n*k$ has been calculated. The computer will have to provide for this, and its provision takes the form of using RAM for temporary storage, so that a number of bytes are reserved for this purpose.

Finally, how does the computer call up the correct subroutine? It's rather simple – when you know. The tokens, like the reserved words, are stored in order in the ROM. Looking for a token means starting at the first one, and checking each in turn, keeping a count. When the matching token is found, the computer will have a count byte (in reserved RAM) of how many places down the table the token was situated. This count number is then added to another address number (stored in ROM), and the result is where the address of the correct subroutine is stored! In practice it's not quite so simple, but the principle is the same – finding an item on one list gives a count number that is then used to find an address on another list.

Chapter Thirty-one

The Miraculous Microchip

In this chapter, we'll get to grips with the Z-80A microprocessor of the Spectrum. The microprocessor, or MPU is, you remember, the 'doing' part of the computer as distinct from the storing part (memory) or the input/output part (the port), so that what the microprocessor does will decide what the computer as a whole does.

The MPU is itself a set of memory stores, but with a lot of organisation added. By means of circuits aptly named *gates*, the way in which bytes are transferred between pieces of memory inside the MPU can be changed and controlled, and it is these actions which constitute the addition, subtraction, logic and other actions of the MPU. Each action is programmed. Nothing will happen unless an instruction byte is present in the form of a 1 or a 0 signal at each of eight terminals, and these bytes are used to control the gates inside the MPU. What makes the system so useful is that because the program instructions are in the form of electrical signals on eight lines, these signals can be changed very rapidly. The speed is decided by another electrical circuit called a 'clock-pulse generator' (or 'clock' for short). The Z-80A can work with a 4 MHz clock, meaning that the clock can be operated at a rate of 4 million pulses per second. The microprocessor will then carry out its internal operations at this rate, but since each complete action may require several internal operations, the rate of carrying out complete machine code instructions is rather less than this – typically a rate of half a million instructions per second. The clock speed that is used by Spectrum is 3.5 MHz–3½ million clock pulses per second.

Hardware and software

The electrical parts of the computer are what we call *hardware*. Changing the design would be hard work, snipping contacts here, soldering new ones there, making it into a mass of spaghetti that the designer would hardly recognise. The program which makes the assembly of hardware work like a computer is a form of *software* (see also Chapter 19). It's a lot more easily altered, because in the case of the Spectrum it is all contained in one ROM chip; change this chip and you have a rather different computer! A few computers in the 'home' category have very small amounts of ROM; practically all of their software is read in from tape or disk and can be changed very easily. This way, if you tire of BASIC you can load in another language and use that instead. Most of us, however, are likely to stay with BASIC in ROM, where it is safe from the effects of ill-judged POKE commands!

As far as the MPU is concerned, software is a collection of bytes, the collection that we call machine code. A program written in machine code is just to our eyes a list of numbers, each one having a value between 0 and 255. Some of these numbers may be instruction bytes, which cause the MPU to do something. Others may be the data bytes, which are numbers to add or store, or which may be ASCII codes. The MPU can't tell which is which, and it is entirely up to the programmer to make sure that everything is done correctly by putting the numbers in the correct order.

The correct order, as far as the MPU is concerned, is quite simple. The first byte that is fed to the MPU after switching on or after completing an instruction is treated as an instruction byte. Now some instructions consist of one byte, and others need to have two or more bytes of further instruction or data following them. If you think of RND and TAB in BASIC, you'll remember that we can use RND by itself (though it can be multiplied by a number), but TAB must be preceded by PRINT and followed by a number in brackets. When the MPU receives an instruction byte, the only way that it can be instructed about what comes next is by the coding of that instruction byte. Instruction bytes therefore come in four types: (a) the ones that are complete in themselves, (b) the ones that need one extra byte of data, (c) the ones that need two extra bytes of data, and (d) some that need an extra instruction byte which may in turn be followed by data. Each instruction byte carries coding that the MPU can use to determine what comes next.

The snag is that the programmer must get it right. 100% right is just about good enough! Feed a microprocessor with an instruction byte when it expects a data byte, or with a data byte when it expects an instruction byte, and you have trouble in a big way. Trouble can mean an endless loop, which causes the screen to stop displaying any new information, and the keys (even BREAK), have no effect so you have to switch off and start all over again. Another possibility is that the machine swings into its switch-on routine, clearing the memory and showing the copyright notice. In either case, you will often lose any stored program (it will always be lost if you have had to switch off) and the moral is that you should always save the program on cassette or Microdrive before you try it out!

What I want to stress at this point is that machine code programming is tedious. It isn't difficult – you are drawing up a set of simple instructions for a simple machine – but it's often difficult for you to remember how much detail is needed. When you program in BASIC, the machine's error messages will help to keep you right and sort out mistakes, but when you write machine code you are on your own, and you have to sort out your own mistakes, though using an *assembler* (see below) helps considerably. Since the best way of learning about machine code is to write and use machine code, and learn from your inevitable mistakes, I'll devote the rest of this chapter to methods of writing machine code numbers before you even learn what the numbers mean and what commands are available, or how to use them. We'll start with ways of writing the numbers that constitute the bytes of a machine code program.

Binary, denary and hex

A machine code program consists of a set of number codes. Since each number is a way of representing the 1's and 0's in a byte of memory, it will consist of numbers between 0 and 255 when we write it in our normal scale-of-ten (denary scale). The program isn't of any use until it can be stored in the memory of the Spectrum, because the microprocessor is a fast device, and the only way of feeding it with bytes as fast as it can cope with them is by storing the bytes in ROM or RAM and letting the microprocessor help itself to them in order. You can't possibly type numbers fast enough to satisfy the microprocessor, and as we saw in Chapter 19, methods like tape or disk just aren't fast enough.

Getting into memory, then, is an essential part of making a machine code program useful, and we shall look at methods in more detail in Chapter 33. At one time, simple and short programs were put into the memory of simple microprocessor systems by the most primitive possible method – having eight switches which could each be set up to give a 0 or a 1 output, and a button which caused the memory to read the binary number that had been set up on the switches. In addition, some method of addressing the memory was needed, and this could be provided by the microprocessor itself, as we shall see later.

Programming like this is just too tedious, however, and working with binary numbers is a process which can cause endless mistakes. Since every binary number is a set of 0's and 1's, after reading and entering a few dozen of them you start to make mistakes, exchanging 0's and 1's, repeating numbers, and all the other possibilities. The obvious step is to make use of the computer itself to place the numbers in the memory, and an equally obvious step is to use a more convenient number scale.

Just what is the most convenient number scale is a matter that depends on how you enter the numbers and how much machine code programming you do. A computer like Spectrum contains subroutines that convert binary numbers into a form that allows it to print denary numbers on the screen automatically, and also carry out the reverse transformation. When you use PEEK, therefore, the address that you use is denary, and the result of the PEEK will be a denary number between 0 and 255. When you use POKE, you can write both the address number and the byte number system that we are used to and which the Spectrum uses for its own PRINT and input commands.

Serious machine code programmers, however, find this just too inflexible a system. By far the best way of entering machine code programs is to write them in what is called *assembly language*, which uses commands that are shortened words. Programs called *assemblers* can be written which will convert these commands into the correct binary codes. So that the programmer need never be bothered with the actual binary codes, many assemblers will show the codes on the screen in a form called hexadecimal, or hex. These assemblers will also require numbers to be typed in using hex code.

Hex codes

Hexadecimal means scale of sixteen, and the reason that it is used so extensively is that it is naturally suited to representing binary bytes. Four bits of binary digits, half a byte, will represent numbers which lie between \emptyset and 15 on our familiar scale, which is the range of just one hex digit (Fig. 31.1). This means that a byte can be represented by two hex digits, and a two-byte address by four hex digits; it is, in addition, much easier to relate the form of a hex number to the pattern of bits in a byte than is possible when you use denary scale. In addition, the number codes that are used as the instruction bytes for microprocessors are generally written in terms of the hex scale and we can see the patterns of organisation in the hex numbers much more clearly, for example, when a set of related commands all start with the same hex digit.

| Hex | Denary |
|---------------|---------------|
| \emptyset | \emptyset |
| 1 | 1 |
| 2 | 2 |
| 3 | 3 |
| 4 | 4 |
| 5 | 5 |
| 6 | 6 |
| 7 | 7 |
| 8 | 8 |
| 9 | 9 |
| A | 1 \emptyset |
| B | 11 |
| C | 12 |
| D | 13 |
| E | 14 |
| F | 15 |
| then | |
| 1 \emptyset | 16 |
| 11 | 17 |
| to | |
| 2 \emptyset | 32 |
| 21 | 33 |
| etc. | |

Fig. 31.1. Hex and denary digits.

Several assemblers are now available for the Spectrum; most can be used with denary numbers, but display the results of their actions in hex. Disassemblers such as the Campbell Software dpas, and the ACS Software INFRARED, show the codes and data numbers in hex, and it is fairly certain that as you progress with machine code you will find that you need to know about hex, even if you avoid using it. You can write your machine code programs for Spectrum entirely in

denary numbers, but you will certainly find that a knowledge of hex will be necessary if you use a different machine. The more advanced books on machine code programming will assume that you are fluent in the use of hex.

The hex scale

The hexadecimal scale consists of sixteen digits, starting conventionally with \emptyset and ascending equally conventionally up as far as 9. The next figure up is not 1 \emptyset , however, because this would mean sixteen (one sixteen and no units), and since we aren't provided with symbols for digits beyond nine, we have to make use of the letters A to F. The number that we write as 1 \emptyset in denary (one ten, no units) is written as \emptyset A in hex, eleven as \emptyset B, twelve as \emptyset C, and so on up to fifteen, which is \emptyset F. The zero doesn't have to be written, but programmers get into the habit of writing a data byte with two digits and an address with four, even if fewer digits are needed. The next number after \emptyset F is 1 \emptyset , sixteen, and the scale then repeats to 1F, thirty-one, which is followed by 2 \emptyset . The maximum size of byte, 255, is in hex terms FF. When we write hex, it is customary to write an 'H' after the code, so that there is no possibility of confusing the hex numbers with denary numbers. A number such as 16 could be sixteen (denary) or twenty-two (one sixteen and six units), but there is no doubt about 16H.

Now the great value of hex is how closely it corresponds to binary code. If you look at the hex-binary table of Fig. 31.2, you can see that 9 is 1 \emptyset 01 in binary and F is 1111. The hex number 9FH is just 1 \emptyset 011111 in binary – you simply write down the binary digits that correspond to the hex digits. The conversion in the

| Hex | Binary |
|-------------|--|
| \emptyset | $\emptyset\emptyset\emptyset\emptyset$ |
| 1 | $\emptyset\emptyset\emptyset 1$ |
| 2 | $\emptyset\emptyset 1\emptyset$ |
| 3 | $\emptyset\emptyset 11$ |
| 4 | $\emptyset 1\emptyset\emptyset$ |
| 5 | $\emptyset 1\emptyset 1$ |
| 6 | $\emptyset 11\emptyset$ |
| 7 | $\emptyset 111$ |
| 8 | $1\emptyset\emptyset\emptyset$ |
| 9 | $1\emptyset\emptyset 1$ |
| A | $1\emptyset 1\emptyset$ |
| B | $1\emptyset 11$ |
| C | $11\emptyset\emptyset$ |
| D | $11\emptyset 1$ |
| E | $111\emptyset$ |
| F | 1111 |

Fig. 31.2. Hex and binary numbers.

opposite direction is just as easy – group the binary digits into fours, starting at the least significant bit (right-hand side), and then convert each group into its corresponding hex digit. Fig. 31.3 shows examples of conversion in each direction, so that you can see how easy it is.

There are a lot of differences between computers in the way that they handle hex numbers. Some, like Spectrum, make no provision at all for the use of hex, assuming that anyone who wants to carry out machine code programming in hex will use an assembler that deals in hex. Others, like the BBC Microcomputer, have a built-in hex translator; the BBC machine even has a built-in assembler.

Assuming for the moment that you do not use an assembler to create your machine code programs, what do you do? The answer is that you design your

Conversion: Hex to Binary

Example: 2CH 2H is $\emptyset\emptyset1\emptyset$ binary
CH is 11 $\emptyset\emptyset$ binary

So 2CH is $\emptyset\emptyset1\emptyset11\emptyset\emptyset$ binary (data byte)

Example: 4A7FH 4H is $\emptyset1\emptyset\emptyset$ binary
AH is 1 $\emptyset1\emptyset$ binary
7H is $\emptyset111$ binary
FH is 1111 binary

So 4A7FH is $\emptyset1\emptyset\emptyset1\emptyset1\emptyset1111111$ binary (an address)

Conversion: Binary to Hex

Example: $\emptyset11\emptyset1\emptyset11$ $\emptyset11\emptyset$ is 6H
1 $\emptyset11$ is BH

So $\emptyset11\emptyset1\emptyset11$ is 6BH

Example: 1 $\emptyset11\emptyset1\emptyset\emptyset1\emptyset\emptyset1\emptyset$... note that this is not a complete number of bytes.

Group into fours, starting with lsb:

$\emptyset\emptyset1\emptyset$ is 2H
1 $\emptyset\emptyset1$ is 9H
11 $\emptyset1$ is DH and
the remaining 1 \emptyset is 2, making 2D92H

Fig. 31.3. Converting between hex and binary.

program in assembly language, which is by far the easiest way to design machine code programs, and then convert into denary code instead of typing the instructions into the assembler program. Converting means looking up, in a set of tables called the *instruction set*, the number which represents each instruction. Instruction sets that are provided by the manufacturers of microprocessor chips or by computer manufacturers are practically always in hex, but the Spectrum manual lists the codes in denary as well, though in numerical order rather than the alphabetical order that we need. Just to assist you, a complete list of Z-80 codes in alphabetical order has been included in this book in Appendix D, using denary, hex and binary forms. Don't refer to it at present – it'll put you off!

Most machine code programs, however, use data bytes as well as instruction bytes, and those are often shown in hex, though it is just as easy to show them in denary. Because of this, it's useful to be able to convert between denary and hex. If you are using the Campbell Software dpas Disassembler, you will then be able to make use of the denary-hex converter program which is part of the package. Otherwise, you will have to convert 'by hand' or use a simple conversion program each time you are working with these codes.

Conversion from hex to denary is illustrated in Fig. 31.4. The method for a one-byte number is very simple – take the denary value of the more significant digit, multiply by 16, and add the value of the other digit. Double-byte numbers such as address numbers are more tedious. If the number is a full four-digit one, the value of the most significant digit is multiplied by $16 \times 16 \times 16$, which is 4096 (all in

Hex to Denary Conversion

(a) Single bytes.

Example: convert 3DH to denary. The value is $3 \times 16 + 13$ (denary D) which is 61 denary.

Example: convert A8 to denary. The value is $10 \times 16 + 8$, which is 168 denary.

(b) Double bytes.

Example: convert 2CA5 to denary. The first digit gives $2 \times 4096 = 8192$. The second digit gives $12 \times 256 = 3072$, and the third digit gives $16 \times 10 = 160$. The last digit is 5, and adding $8192 + 3072 + 160 + 5$ gives 11429.

Example: convert F3DBH to denary.
1st digit .. $15 \times 4096 = 61440$
2nd digit ... $3 \times 256 = 768$
3rd digit .. $13 \times 16 = 208$
4th digit .. 11 = 11
Sum = 62427

Fig. 31.4. Hex to denary conversion, single or double byte.

denary). This value is noted, and the value of the next digit along is multiplied by 256, and also noted. The next digit value is then multiplied by 16, the result noted, and the least significant digit value is also written down. Finally, all of these values are added to get the final denary figure. Examples are shown in Fig. 31.4.

Denary to hex is not so simple. A single-byte number is dealt with by division by 16, which will give a whole number part (the integer part) and a fraction. The whole number is converted into a hex digit, and the fractional part by itself is multiplied by 16, and the result converted into another hex digit. For address numbers, division by 16 will result in an integer part which is greater than 16. Convert the fractional part into a hex digit by multiplying the fraction by 16, and then treat the integer part in the same way again, dividing by 16, writing down the integer, and converting the fractional part into a hex digit. Continue until the integer part is less than 16, and then write down its hex value. This method finds the digits in order starting at the least significant digit. Fig. 31.5 shows some examples of the conversion.

Denary to Hex Conversion

(a) Single bytes - numbers less than 256 denary.

Example: convert 153 to hex.

$153/16 = 9.5625$ so 9 is upper digit, lower digit is $0.5625 * 16 = 9$, so that number is 99H.

Example: convert 58 to hex.

$58/16 = 3.625$, so 3 is upper digit, lower digit is $0.625 * 16 = 10$, A in hex. So that number is 3AH.

(b) Double bytes - number between 256 and 65535.

Example: convert 23815 to hex.

$23815/16 = 1488.4375$. $0.4375 * 16 = 7$, lowest digit.

$1488/16 = 93$, 0 remainder - 0 is next digit.

$93/16 = 5.8125$. $0.8125 * 16 = 13$, hex D. Last digit is 5, so that the complete number is 5D07H.

Fig. 31.5. Denary to hex conversion, single or double byte.

It's simpler to use programs for the conversion. Denary to hex can be done very simply by successive divisions by 4096, 256, and 16, converting the whole number part of the answer to hex digits each time. The conversion is done by using ASCII codes, because when your Spectrum prints a number in hex, it must be printed as a string, because it includes letters as well as number digits. As it happens, there is a fairly simple relationship between the ASCII codes for digits 0 to 9, and the numbers themselves. If you add 48 to the number, then you have the ASCII code

for that digit, which can be printed as a string character. A slight complication arises when we get to ten, because in hex this is A, and the ASCII code for A is 65, which is 55 greater than 10. The conversion program must therefore add 48 to a digit of 9 or less and 55 to a digit of ten to fifteen to make the correct conversion to hex code. This isn't difficult for a BASIC program, and an example of denary to hex conversion program is shown in Fig. 31.6.

```

10 CLS: PRINT "Please type denary number_":
20 INPUT d
30 IF d > 65535 THEN PRINT " Too large - limit
is 65535": PAUSE 50 : GOTO 10
40 IF d < 1 THEN PRINT "No numbers less than
unity, please": PAUSE 50: GOTO 10
50 IF d <> INT d THEN PRINT "No fractions,
please": PAUSE 50: GOTO 10
50 LET f = 4096: LET h$ = ""
60 LET y = INT (d/f)
70 GO SUB 200
80 LET d = d - y * f : LET f = INT (f/16)
90 IF f < 1 THEN GOTO 110
100 GOTO 60
110 FOR n = 1 TO 3: IF h$(1) = "0" THEN LET h$ =
= h$(2 TO )
120 NEXT n
130 PRINT "Hex number is "; h$ + "H"
140 GOTO 9999
200 IF y <= 9 THEN LET h$ = h$ + CHR$(y + 48)
210 IF y > 9 THEN LET h$ = h$ + CHR$(y + 55)
220 RETURN

```

Fig. 31.6. A BASIC program for denary to hex conversion.

Hex to denary can be done in much the same way, converting the ASCII code for each digit into the number digit itself, multiplying for the correct place factor (16,256,4096), and then adding. The conversion can use a loop for both the multiplication and the addition, to obtain the denary number. Once again, the BASIC program is fairly simple (see Fig. 31.7) which is why the '£5 per published letter' pages of magazines are always choked with denary-hex conversion programs for each new computer.

Throughout this section, we shall use mainly denary codes, with a few hex values thrown in where they are needed. It's only fair to point out, though, that there are parts of machine code programming which do need some working at in terms of mastering arithmetic methods. The most important of these is the way of representing negative numbers.

```

100 CLS: LET y = 1 : LET d = 0 : PRINT "Please
type hex number" : INPUT h$
200 IF LEN h$ > 4 THEN PRINT "Too long - maximum
of four \"\" characters please": PAUSE 100 :
GOTO 100
300 LET p$ = h$ (LEN h$): LET h$ = h$ (1 TO
(LEN h$ - 1))
500 GO SUB 200 : IF LEN h$ > 0 THEN GOTO 300
600 PRINT "Denary number is "; d
1000 GOTO 9999
2000 LET a = CODE p$
2100 IF a < 48 OR a > 102 THEN GO SUB 3000
2200 IF a < 65 AND a > 57 THEN GO SUB 3000
2250 IF a <= 97 AND a > 70 THEN GO SUB 3000
2300 IF a <= 57 THEN LET q = a - 48
2400 IF a >= 65 THEN LET q = a - 55
2500 IF a >= 97 THEN LET q = a - 87
2600 LET d = d + q * y : LET y = y * 16
2700 RETURN
3000 PRINT "Bad hex ... please try again ":
PAUSE 100: RETURN

```

Fig. 31.7. A BASIC program for hex to denary conversion.

Negative numbers

We represent negative numbers in denary by the use of a negative sign, so that we can write values like +15 and -15, using the same digits but with the signs + and - showing whether the values are positive or negative. Microprocessors make no provision for the use of these signs, so that binary code has to use one bit from a byte to represent the sign, and the bit that is always chosen is the most significant bit (left-hand side). The convention that is followed is that if the most significant bit is a 1, then the sign of the byte is negative, and if the most significant bit is a 0, then the sign is positive. It's a simple convention, and as it happens a particularly useful one, but it does have disadvantages for human operators. One of these disadvantages is that the digits for a negative number are *not* the same as those of its positive counterpart: +5 in binary is 00000101 but -5 is 11111011, which doesn't look like the same number. A second disadvantage is that using one bit as a sign bit leaves fewer bits for representing the number value. If we use the highest bit of a single byte number as a sign bit, then the remaining 7 bits can represent numbers only as high as +127. We can, of course, also represent a negative number down to -128, so that we haven't lost anything from the range of numbers that we can represent. If we use two bytes, then losing one bit to use as a sign bit means that the range available is -32768 to +32767 in denary. Using five bytes for each number, as Spectrum does, means that a single bit used for sign purposes does not greatly affect the range of numbers that we can use.

The third disadvantage is that human readers cannot distinguish between a single byte number which is negative, and one which is written with no regard for sign. The short answer is that the human operator doesn't need to worry - the microprocessor will use the number in the same way no matter how we happen to think of it and, in this section, you will soon see how the conversion is made and used and some practice will make you completely confident with the methods.

Most of our applications of negative numbers call for single-byte numbers to be used, so we shall look at conversions to negative form for just this range. The binary number conversions are the basis of all the others, so we shall look at them first, though we may not use them to any great extent. To start with, we can't form a negative version of a single-byte number whose denary value is more than +127, or negative value is less than -128, because such numbers require more than one byte. The positive version of the number is written in eight-bit form, and this may mean padding it out with some 0's on the left-hand side. If the most significant bit is not zero, the number won't convert.

Each bit is then inverted. This means writing a 1 in place of each 0 and a 0 in place of each 1, as illustrated in Fig. 31.8. The resulting number, called the complement, has 1 added to it, and this gives the negative form or 'two's complement' as it is also known. The illustration in Fig. 31.8 shows the process in action, and bears out the point that the binary number in negative sign form is nothing like its positive counterpart. Note the process of binary addition, where $1+0=1$, $1+1=0$ and carry 1, and $1+1+$ carry 1=1 and carry 1.

| | | |
|---------------|---------------|------------|
| Binary number | 00110110 | Denary 54 |
| inverted | 11001001 | |
| add 1 | 11001010 | Denary -54 |

| | |
|-------------------|------------------------------|
| denary number | -5 |
| In binary this is | 101, and in eight-bit binary |
| | is 00000101 |
| Inverted, | this is ... 11111010 |
| Add 1 | 11111011 which is byte |
| | for -5 |

Fig. 31.8. Forming the two's complement (negative form) of a binary number.

What it amounts to in denary terms is that a single-byte number ranging between 128 and 255 is negative, and numbers of 127 and less are positive. To find the equivalent value of a negative number in denary terms, simply subtract the value of the number from 256 (Fig. 31.9 shows examples). This is by far the easiest way to handle these negative numbers in the only point where we have to, which is in jump-relative instructions, dealt with later.

The hex equivalent of negative binary numbers can be found from either the binary or the denary versions. Some microprocessor actions will treat any number greater than 127 denary as a negative number (7FH), causing the equivalent of a

| | |
|------------------|---|
| Denary number -5 | Equivalent byte, in denary, is $256 - 5 = 251$ |
| Denary number -8 | Equivalent byte, in denary, is $256 - 8 = 248$ |

Fig. 31.9. The denary equivalent of single-byte negative binary numbers.

subtraction when it is added to any other number. Other microprocessor actions will treat all numbers as unsigned, meaning that no importance will be attached to the sign bit which will be counted as a number of 128's. The only problem arises when you are trying to find out what the microprocessor has done. In general, when you read in an instruction set that a number is treated as 'signed', this indicates that the most significant bit will be treated as a sign bit. When the number is said to be 'unsigned', it will be treated simply as a binary number, with a byte able to represent positive numbers between \emptyset and 255.

Chapter Thirty-two

Registers, Addresses, and Assembly Language

Registers – PC and accumulator

A microprocessor consists of sets of memories, of a rather different type compared with ROM or RAM, which are called *registers*. These registers are connected to each other and to the pins on the body of the MPU by the circuits called gates. In this chapter, we shall look at some of the most important registers in the Z-80 (identical to the Z-80A and Z-80B) and how they are used. A good starting point is the register called the PC (or *Program Counter*).

The PC is a sixteen-bit register which can store a full-sized address number, up to FFFFH or 65535 denary. Its purpose is to count instruction bytes (not programs!), so that the number that is contained in the register will be incremented (increased by 1) automatically each time an instruction byte is read. This is a completely automatic action which doesn't need any instructions from the programmer, because it's built into the action of the Z-80. The PC register will start at a count of zero when the Z-80 is first switched on, so this is where the first instruction of the ROM will start.

The usefulness of the PC is that it is the method by which memory is addressed. When the PC contains an address, the electrical signals corresponding to the \emptyset 's and 1's of that address appear on a set of connections, collectively called the address bus, which link the microprocessor to all of the memory, RAM and ROM. The number which is stored in the PC register therefore selects one byte in the memory, the byte which has that address. At the start of an instruction, the microprocessor will send out a signal called the *read signal* on another line, which will cause the memory to connect its stored bits to another set of lines, the *data bus*. The signals on the data bus therefore correspond to the pattern of \emptyset 's and 1's stored in the byte of memory that has been selected by the address in the PC. Each time the number in the PC changes, another byte of memory is selected, so this is the way that the microprocessor can keep itself fed with bytes, incrementing the number in the PC each time a byte has been read.

There are other ways in which the PC number can be changed, but for the moment we'll pass over that and look at another register, the accumulator. The accumulator is the main 'doing' register of the MPU, meaning that you would normally fill it by copying a byte from memory (loading the accumulator), or use it to write to memory (loading memory from the accumulator). The memory byte which is used in each case will be the one whose address is stored in the PC at the time.

As the name suggests, the accumulator also holds the result of operations. If you have a number byte stored in the accumulator, you can add another number to it, and the result will be stored back in the accumulator. It's as if you had a number variable called *Total*, and you wrote the BASIC line:

LET *Total* = *Total* + extra

where *extra* is a number that you add to *Total*. The difference, and it's an important difference, is that the accumulator can't store a number greater than 255 because it is a single-byte register.

The importance of the accumulator is that there are more instructions which affect or use this register than any of the many other registers in the Z-80. When a byte is read into the MPU, it is generally read into the accumulator. When arithmetic is done it is normally done in the accumulator. When a byte is stored in memory it is usually stored from the accumulator. Unlike earlier designs of microprocessors, the Z-80 has a large number of registers which can be used in much the same way as the accumulator, but none of them has the same range of possible operations.

Addressing methods

When we program in BASIC, we don't have to worry about memory addresses – these are taken care of by the operating system in the ROM. When a variable is allocated a value in a BASIC program as, for example, by a line like:

LET *n* = 12

we never have to worry about where the number 12 is stored. Similarly, when we write:

20 LET *k* = *n*

we don't have to worry about where the value of *n* was stored so as to copy it into *k*. Remembering our comparison with wall building back in Chapter 19, we can expect that when we carry out machine code programming we shall have to specify each number that we use or alternatively the address at which the number is stored. This latter is referred to as the addressing method. What makes it particularly important is that a different code number is needed for each different addressing method used for each command. This means that each command exists in a number of versions, one code for each possible addressing method. What we shall do here is to look at examples of some addressing methods and the way that we indicate them in assembly language.

Assembly language

Trying to write down machine code directly as a set of numbers is a very difficult and error-prone activity. The most useful way of starting to write a program is to write it as a set of steps in what is called assembly language (or assembler language), which is a set of abbreviated words called *mnemonics*, and numbers

which will be data or address numbers. The numbers can be in hex or in denary. Each line of an assembly language program indicates one microprocessor action, and this set of brief instructions is later 'assembled' into machine code, hence the name.

The aim of each line of an assembly language program is to show the action and the data or address that is needed to carry out that action, just as when we program a TAB in BASIC we need to complete it with a number. The part of the assembly language that specifies what is done is called the *operator*, and the part which specifies what the action is done to or on is called the *operand*. A few instructions need no operand, and we'll look at some of them later, but for the most part, the operand for Z-80 instructions consists of two parts, one which specifies a register and another which specifies a data byte or an address.

An example makes this easier to follow. Suppose we look at the assembly language line:

LD A,12

The operator is LD, a shortened version of LOAD, which is used for each transfer or copying of a byte from one address to another or one register to anywhere else. A is the first part of the operand, and the abbreviation means *Accumulator*. Its placing immediately after the operator means that this is a destination for a byte, the place where the byte will end up at the completion of the instruction. The other part of the operand, following the comma, is the number 12. Generally, when the number is written in this way it means denary 12 rather than hex 12, which would be typed as 12H.

The whole line, then, should have the effect of placing the number 12 into the accumulator register of the Z-80. It is the equivalent in machine code terms of the BASIC command:

LET *a* = 12

if we could imagine that the number variable 'a' existed as a hardware circuit inside the MPU, rather than as an entry in the memory which we know it is.

A command such as LD A, 12 is said to use *immediate addressing*, because the byte which is loaded into the accumulator is placed in the memory address which immediately follows the operator code. There is one single byte code for the LD A part of the line, and this byte is 62 (3EH), so that the sequence 62, 12 in memory will represent the entire command LD A,12. It's a lot easier to remember what LD A,12 means than to interpret 62,12, however, which is why we use assembly language as much as possible.

Immediate addressing can be convenient, but it ties you down to the use of a definite number, like programming in BASIC:

LET *n* = 4*12 + 3

rather than

LET *n* = *a***b* + *c*

In the first example, n can never be anything but 51, and we might just as well have written:

LET n = 51

The second example is very much more flexible, and the value of n depends on what we choose for the variables a, b and c. When a machine code program is held in RAM, then the numbers which are loaded by this immediate addressing method can be changed if we want them changed, but when the program is held in ROM no change is possible, and that's just one reason for needing other addressing methods. One of these other methods is *direct addressing*.

Direct addressing uses a complete two-byte address in the operand. This creates a lot of work for the Z-80, because when it has read the code for the operator and the first part of the operand (one byte of code) it will then have to read the next two bytes immediately following the instruction byte, place these two bytes of address in the PC, read in the data byte from this address, deal with it, and then restore the PC address to its correct value (Fig. 32.1). A direct operation is therefore slow, and needs a lot of bytes of memory to store all the bytes that are needed.

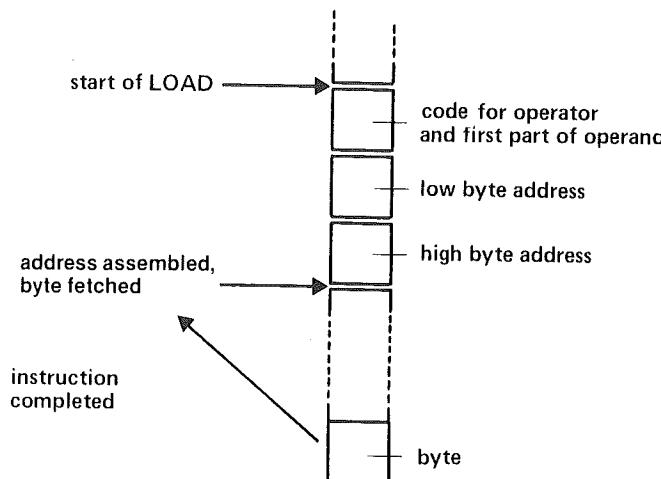


Fig. 32.1. How the bytes of a direct-addressed command are stored in the memory.

Suppose, for example, that we have the instruction:

LD A,(7FFFH)

which appears in lists as LD A,(NN), where NN means a full address. In assembly language, the operator is LD, load, and the destination for the byte is the accumulator A. The source of the byte is the address 7FFFH (denary 32767), and the assembly language uses the brackets to mean 'contents of'. It's a way of reminding yourself that what is put into the A register is not 7FFFH (which wouldn't fit) but *the byte which is stored at this address*. The effect of the complete instruction, then, is to place a copy of the byte stored at address 7FFFH into the

accumulator. When the instruction has been completed, the address 7FFFH will still hold its own copy of the byte because reading a memory does not change the content of memory in any way.

The way in which operands are written in assembly language follows the order destination, source, so that if we write:

LD (7FFFH),A

then this means that the byte stored in the accumulator is copied to the address 7FFFH. Note the use of the brackets again. Some types of microprocessors use the word ST (store) for this type of action, but the Z-80 uses only the order of writing the quantities and symbols.

Indirect addressing

Immediate addressing and direct addressing (also called extended addressing) are useful, but the Z-80 also permits a very handy form of what is called *indirect addressing*. Indirect addressing means going to an address, an address pair in fact, to find another address, and then using this second address to find or send the data byte. It's rather like going to a travel agency to find the address of a hotel in which you can take a room (or eat a bite?). The form of indirect addressing which the Z-80 uses is called register-indirect, and it makes use of some of the other registers in pairs.

The use of eight-bit registers in pairs to hold a full sixteen-bit address is a special feature of the Z-80, and one which helps to make it such a popular microprocessor with designers of computers that are intended for business and other serious uses. There are three sets of such registers which are labelled as HL, BC and DE respectively and, of these, the HL pair are the ones used most frequently for this purpose. All of these registers can be used singly, incidentally, just as if they were spare accumulators, but with a more limited range of actions.

We can load a complete address into the HL pair of registers by using a command which is written in assembly language in the form:

LD HL,32767 or LD HL,7FFFH

This means that the high byte of the address, 7FH in the hex version, will be held in the H register (H for high), and the low byte FFH will be stored in the L register (L for low). The names, as you can see, have been picked deliberately to remind you of which byte is stored where. We can then load the accumulator (or another register) with the byte which is stored at this address 7FFFH by using the command:

LD A,(HL)

or we can store a byte which is in the accumulator to the address 7FFFH by using:

LD (HL),A

This is another example of how the order of writing the operands is used to

indicate which is source and which is destination; the brackets have also been used in their usual meaning of 'contents'.

Now you might think that this is just a rather long-winded way of writing the command LD A,(7FFFH), but there is an important difference. Once an address number has been placed in HL, we can increment or decrement that address with a single-byte (no operand) instruction. If we have loaded HL with the address 7FFFH, then the instruction:

DEC HL

will cause the address number stored in HL to become 7FFEH (denary from 32767 to 32766), so that if we use:

LD A,(HL)

again, the accumulator is loaded from address 7FFEH (32766) rather than from 7FFFH (32767). If you compare this with a loop in BASIC, then you can see how instructions like this can be used in a loop to allow a different address to be used on each pass through the loop, decrementing HL on each pass round the loop as in this example or, of course, incrementing if this is what is needed.

PC-relative addressing

PC-relative addressing is one of the simplest and most primitive methods of obtaining an address in the PC. The operand contains a byte called the displacement, which is added to the address which is stored in the PC of the microprocessor, and the resulting number is then the address which is used for the load, store or whatever is being carried out. Early types of microprocessors used PC-relative addressing for practically all of their actions, but the Z-80 uses it only for one small set of instructions – the jump-relative (JR) instructions.

A jump-relative means a transfer to a new address using a PC-relative addressing method. The complete JR instruction consists of two bytes, the operator JR and the operand, which consists of a condition and the displacement. The instruction needs some care and experience, though, because the displacement byte is treated as a signed byte, meaning that if the most significant bit of the byte is 1 (denary value 128 or more) then the byte is treated as a negative number, and the address which is obtained when this byte is added will be *lower* than the address which existed in the PC before the JR instruction. In terms of denary numbers, then, if the byte is 127 or less, there will be a jump forward; if the byte is 128 to 255, there will be a jump backward. When the jump of address has occurred, the PC will then resume normal action from that address and will not return to its previous address unless by the action of incrementing (if the jump was back) or by another jump (if the jump was forward).

When we use PC-relative addressing, we will have to work out the value of the displacement byte. When an assembler program is used to convert the assembly language into code, the displacement byte will be calculated by the assembler

program, but when you assemble 'by hand', then you will have to calculate it for yourself. The calculation is like this:

- (1) Write down the address at which the operator byte of the JR instruction will be placed. This is the source address.
- (2) Write down the address to which the program must jump, which is the destination address.
- (3) Subtract source address from destination address, and then subtract two from this answer. What you now have is the displacement in denary terms. If it is positive, use it directly. If it is a negative number, subtract the value from 256, and use the result as the displacement.

Why subtract 2? It's because the displacement is always calculated from the operator address, but the jump can't take place until the operand containing the displacement has been read (which means that the PC will have incremented), and in addition, the PC will increment automatically at the end of the instruction. By subtracting 2, we allow for these two incrementing actions, and obtain the correct displacement byte. Fig. 32.2 shows some examples of positive and negative displacements. It's often easier, if you haven't decided on address numbers, just to count bytes between source and destination. Note that PC-relative addresses cannot cause a jump to more than 127 places forward, or 128 places back from the PC address of the operator, because this is the complete range of a signed byte.

| | |
|---------------------------|---------------------------------------|
| Source address 32542 | Difference is 23 |
| Destination address 32565 | Subtract 2 to get 21 |
| | 21 is displacement number |
| Source address 32533 | Difference is -29 |
| Destination address 32504 | Subtract 2 to get -31 |
| | 256 - 31 = 225 is displacement number |

Fig. 32.2. Positive and negative displacements for a program-counter relative addressed instruction.

The other Z-80 registers

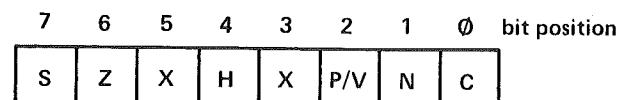
We've mentioned a number of the Z-80 registers already. The PC is the addressing register, which keeps a count of the address of each instruction byte and data byte of the program – it's the 'where-are-we-now' register. When a machine code program is run, the address of the first byte of the program must be placed into the PC, and the microprocessor will then take over quite automatically, so we need some method of placing an address into the PC. We'll look at the Spectrum

methods of doing this later, but generally we leave the PC alone once the program has started.

The accumulator is the register in which most of the work is done, and we'll look at some of the accumulator operations in detail in the next chapter. There are six other single-byte registers, labelled as B,C,D,E,H and L which can be used rather as we use the accumulator itself, though none of them offers quite such a large range of actions. In addition, as we have seen, these registers can be grouped as BC,DE and HL to store complete 16-bit numbers, such as addresses. A limited number of 16-bit arithmetic operations are also possible in the HL register pair.

This does not exhaust the register count of the Z-80. There are two registers which we seldom, if ever, use – the interrupt (I) and the refresh (R) registers whose uses are rather specialised. Another single byte register, however, the *status or flag register*, is of very great importance despite the fact that we cannot store or load it directly. The status or flag register is a way of keeping a score of results. When an arithmetic operation like addition or subtraction, or a logic operation like AND, OR or XOR (see Chapter 19), is carried out, the result in the accumulator may be a number that is positive, negative or zero. This 'status' of positive, negative or zero is indicated by the state (\emptyset or 1) of bits in the *status register*. The status register is not particularly well named, because it's not really a register which can store a number, but just a collection of single bit stores with no connection between them. Some books call this the *flag register* because this is such a descriptive name – a flag is raised each time one of these results is available, and stays raised until a new result appears.

Fig. 32.3 shows the layout of the Z-80 status register. The bits that we will be most interested in are the S, Z and C flags, bits 7, 6 and \emptyset respectively. The S-flag is 1 when the byte in the accumulator is negative after an arithmetic/logic operation; the flag value will be \emptyset if the accumulator byte is positive or zero. The Z-flag is 1 if the accumulator contains zero after any of the operations that affect the flags but it will be \emptyset otherwise. The C flag is 1 if there has been a carry resulting from an addition, or if a borrow is needed in a subtraction. Any register which can be used for the operations that affect the flags will also be signalled by the flags, so that if



Carry flag—set if there is a carry or borrow in arithmetic
 N-Flag—set for subtract operation
 Z-Flag—set by zero result of some operations
 S-Sign flag—set if result is negative

(Other flags have rather specialised uses)

Fig. 32.3. The Z-80 status register. Bits 3 and 5 are not used (they may be either 0 or 1 permanently).

you are working with the C registers and a subtraction causes a zero to appear, this will be signalled by the Z-flag in the status register just as if the action had taken place in the accumulator.

The status register can't normally be affected by the programmer, because it exists to signal precisely the results of actions. The programmer very seldom knows directly what is stored in the status register, and its importance lies in the fact that it controls jumps. To make a comparison with BASIC again, suppose we programmed in BASIC:

100 IF a = \emptyset THEN GOTO 300

where a 'jump' to line 300 is carried out if the value assigned to variable a happens to be zero. We may not know at that instant what the value of a will be, only that the jump will take place when a is zero. The machine code version of this, in assembly language, is:

JR Z,Disp

where JR is the jump-relative operator, Z is the part of the operand which indicates what flag is used as a condition for the jump, and Disp means the single-byte displacement number which will direct the jump, if it is taken, to the correct address. When this action is carried out by the microprocessor, the single byte of code that represents the JR Z part of the command causes the microprocessor to check the status of the Z-flag in the status register. If the previous arithmetic/logic action left the accumulator (or whatever register was being used) at zero, then the Z-flag will be set (meaning it will be a 1 bit), and the displacement byte will be added to the PC address to cause the jump to be carried out. If the Z-flag is not set (reset, at \emptyset), then the displacement is ignored, just as the instructions following THEN in BASIC are ignored when the condition is not met. In the BASIC example, if variable a is not zero, the program does not GOTO 300. In the machine code example, the next address held in the PC will be the one (Fig. 32.4) which follows the address of the displacement byte rather than the PC + displacement address that we would find if the condition was met by having the accumulator (or other register) zero.

One peculiarity of the way in which the Z-80 uses its status register is that only certain actions, particularly arithmetic and logic actions, actually affect flags. Load and store operations do not affect flags, so if you have previously used 6502

Address - JR Z, disp

Z-flag set - jump to new address given by address of JR instruction + displacement + 2

Z-flag not set - ignore displacement and move to instruction following JR

Fig. 32.4. A conditional jump instruction.

machine code, you will have to adjust your thinking! For example, suppose we have a piece of assembly language that reads:

```
LD A,(HL)
DEC A
LD A,(DE)
JR Z,Disp
```

If the DEC A (decrement the accumulator) action, which is one that can affect flags, causes the contents of the accumulator to become zero, then the jump at the JR Z,Disp stage will be carried out, even though the accumulator has been reloaded from DE and probably does not contain a zero byte any longer. This is very much a Z-80 peculiarity, and one that can at times be very useful.

Index registers

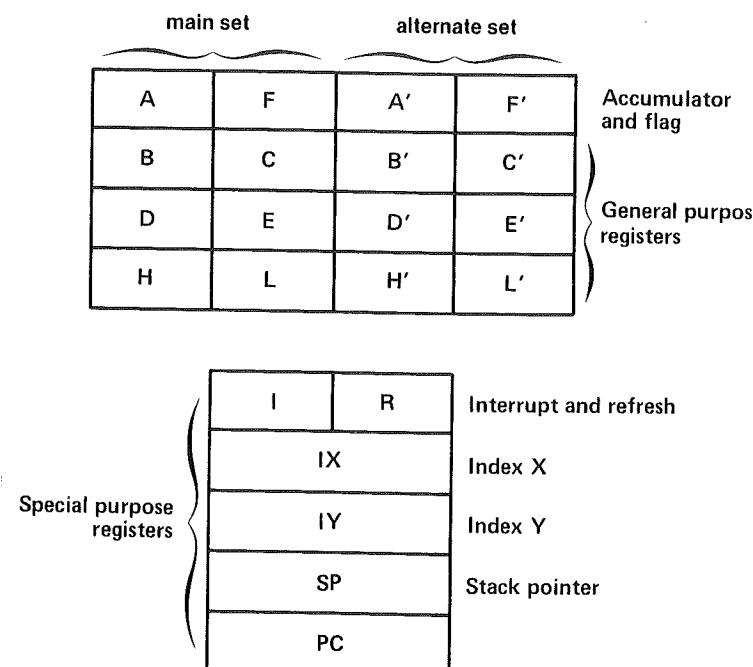
The Z-80 also contains two index registers. These are 16-bit registers, so that they can hold a full 16-bit address number, and they are labelled as IX and IY. The reason for the word 'index' is that these registers can be used in a form of addressing that is similar to the action of a book index. An address called the *base*

address or *page address* is held in an index register, and any address up to 127 numbers higher or 128 numbers lower than this base address can be used by adding a displacement byte to the index address. In assembly language, this is written as $(IX + d)$ or $(IY + d)$, where d means the single byte displacement. We can use commands such as:

```
LD A,(IX + d)
```

in this way, meaning that the accumulator will be loaded from the address which is equal to the base address stored in IX plus the displacement. The IX register would have to have been loaded earlier in the program. We shall not elaborate on this very brief description, because the IX and IY registers are used very extensively in the operating system of the Spectrum, and the manual cautions against their use in machine code programs.

Figure 32.5 shows a 'map' of the Z-80 registers to give an overall view of what is available. As well as the main set of A,F,B,C,D,E,H and L registers, there is also an 'alternate set' which can be used. The alternate registers are labelled as A',F',B',C',D',E',H', and L'. These can be selected to be used in place of the main set by register exchange commands – you can't use the main and the alternate registers at the same time.



The main and alternate registers can be exchanged by instructions such as EX AF, A F' and EXX.

Fig. 32.5. The Z-80 registers. The alternate set cannot be used at the same time as the main set.

Chapter Thirty-three

What the Registers Do

Accumulator actions

Since the accumulator is the main single-byte register, we can list its actions, and describe them in detail, knowing that the description will also hold true for any of the other single-byte registers that can be used in the same way. Of all the single-byte register actions, simple transfers of bytes are by far the most important. We don't, for example, carry out any form of arithmetic on ASCII code numbers, so that the main actions that these require of the microprocessor are fetching and storing – loading the accumulator from one memory address and storing the byte back at another address. The systems in which the Z-80 is used do not permit a byte to be copied directly from one memory address to another, so that the rather clumsy-looking method of loading from one address and storing to another is used almost exclusively.

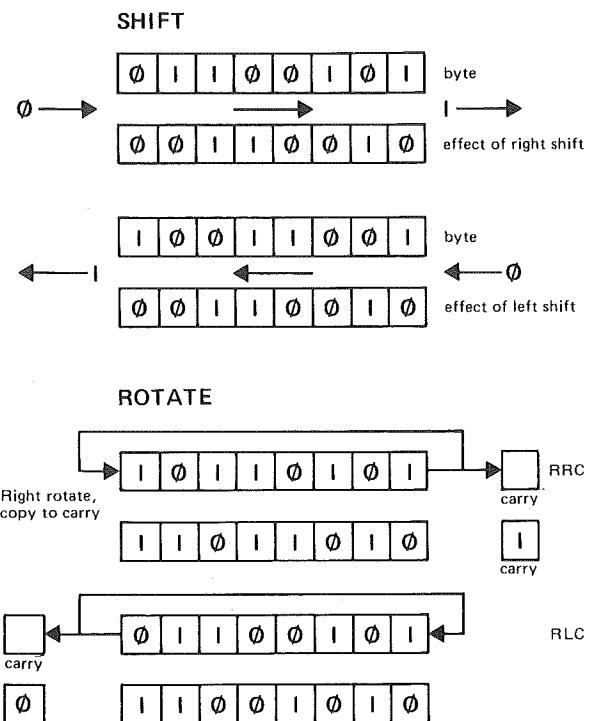


Fig. 33.1. The effect of SHIFT and ROTATE commands. This has been simplified – the Z-80 SHIFT and ROTATE commands often involve the bit called the carry flag as well as the register contents.

The next most important group of actions is the arithmetic and logic group, which contain addition, subtraction, AND, OR and XOR and CP. We can add two others to this group: SHIFT, which causes all the bits of a byte to move along one place (you have to specify in the command whether you want left shift or right shift, among other details), and ROTATE, which connects the ends of the register before carrying out a shift (see Fig. 33.1).

The effect of the main shift and rotate commands, with their assembly language mnemonics, is shown in Fig. 33.2. A shift always results in the register losing one of its stored bits, the one at the end which is shifted out, and gaining a byte, which may be zero, at the other end. A rotation, by contrast, keeps the bits stored in the register, but changes the position of the bits in the byte, though preserving the relative order. These instructions are used mainly for selecting half-a-byte (one hex digit) for arithmetic purposes, or for sending one bit at a time out from, or reading one bit at a time into the register as is needed in cassette recording or replaying operations. Fig. 33.3 summarises the arithmetic and logic group of commands, and their assembly language mnemonics.

A third group is comprised of increment, decrement, and comparison. Increment and decrement mean adding and subtracting, respectively, the number one; and when they are applied to a single register, mean that the number stored in that register will be incremented or decremented. When one of the double registers is used, then INC and DEC can be used in two ways. The command INC HL, for example, will increment the number that is stored in the HL register pair, just as INC A will increment the number stored in the A register. The command INC (HL) will, however, increment the byte which is stored in the address held in the HL pair, without any effect on the address itself, so that this single instruction can replace the set:

```
LD A,(HL) ; put byte into accumulator
INC A      ; increment the byte
LD (HL),A  ; put it back at the same address
```

Note the way that comments are put into the assembly language statements, treating the semicolon like a REM in BASIC – anything following the semicolon is a comment and not part of the instruction.

When a single register is incremented or decremented, the result will affect the flags in the status register, so that the S-flag will be set (to 1) if the result is negative, reset (to 0) if the result is positive or zero, and the Z-flag will be set (to 1) if the result is zero. When the byte whose address is held by a register pair is incremented or decremented using INC(HL) or DEC(HL), the same applies; flags will be set or reset according to the result of the action. When a double register pair is incremented or decremented, however, by such instructions as INC HL or DEC HL, then the status register is *not* affected. This is a quirk of the instruction set of the Z-80 which is often rather a nuisance, but there isn't much we can do about it. Ways of programming so as to set flags will be dealt with later.

CP, the mnemonic for compare, is a particularly useful form of arithmetic instruction. CP has to be followed by a byte, a register, or some source of a byte

| | |
|-------------|--|
| SLA | Left shift, with MSB shifted to carry flag |
| SRA | Right shift, MSB not changed, LSB to carry flag |
| SRL | Right shift, zero to MSB, LSB to carry flag |
| RLD and RRD | not used to any extent |
| RLCA | Left rotation of accumulator, bit 7 copied to carry flag |
| RLA | Left rotation of accumulator, including carry |
| RRCA | Right rotation of accumulator, LSB copied to carry |
| RRA | Right rotation of accumulator, including carry |
| RLC | Left rotation of register, MSB copied to carry |
| RL | Left rotation of register, including carry |
| RRC | Right rotation of register, LSB copied to carry |
| RR | Right rotation of register, including carry |

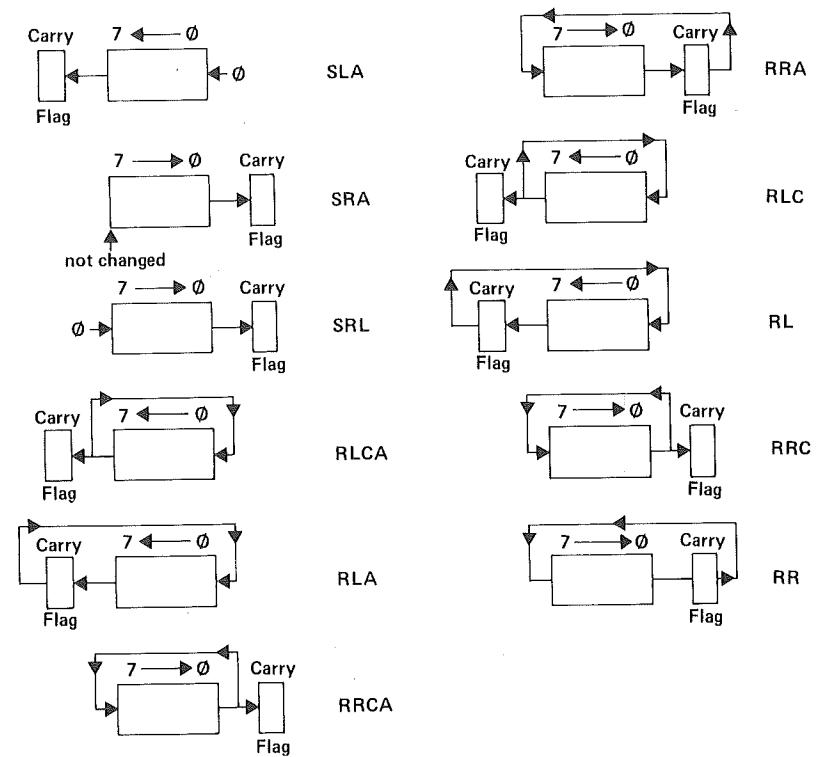


Fig. 33.2. The Z-80 main SHIFT and ROTATE commands, with their effects and the mnemonic codes.

| | |
|------------|---|
| ADD A, r | Add the byte in the register r to the byte in the accumulator. Store result in accumulator, with any carry used to set the carry flag of status register. |
| ADC A, r | Add the byte in the register to the byte in the accumulator, and add in any carry, as indicated by the carry flag. Store result in the accumulator, and if there is another carry, then set the carry flag, otherwise reset the flag. |
| SUB r | Subtract the byte from the byte in the accumulator, and store the result back in the accumulator. Set the carry flag if there has been a borrow. |
| SBC A, r | Subtract the byte in a register, and the carry bit also, from the byte in the accumulator. Store the result back into the accumulator, and set the carry flag if there has been another borrow. |
| AND r | AND the byte represented by r with the byte in the accumulator, store the result in the accumulator. Affects S and Z flags mainly. |
| OR r | OR the byte with the byte in the accumulator, and store the result in the accumulator. Affects S and Z flags mainly. |
| XOR r | XOR the byte with the byte in the accumulator, and store the result in the accumulator. Affects S and Z flags mainly. |
| ADD HL, rr | Add contents of register pair to contents of HL pair. Result stored in HL pair. Set carry flag if there is a carry from the MSB of H. |
| ADC HL, rr | Add with carry bit the byte in a register pair to the bytes in HL. Result stored in HL. Carry flag set if there is a carry out. |

SBC HL, rr Subtract the contents of the register, and the carry bit, from the contents of HL. Store result in HL, set carry if a borrow takes place.

NOTE r has been used to indicate a register, such as B, C, D, E, etc., or an immediate-loaded byte, or the contents of an address or an address held in a register pair, such as (HL). Not all addressing methods will be available for each command.

Fig. 33.3. The arithmetic and logic group of instructions – samples only!

such as (HL) or (address), and it compares the byte that is fetched with the byte that already exists in the accumulator. Comparing is rather like subtraction, but there is one vital difference. If, for example, we had the byte 50 in the accumulator, then SUB 40 would have the effect of leaving 10 in the accumulator, SUB 50 would leave zero in the accumulator, and set the zero flag in the status register. CP 40 would do nothing very noticeable, and CP 50 would set the zero flag, but neither of these CP instructions would alter the value of the byte in the accumulator. CP allows us to find out what a subtraction would do without altering the accumulator, and we can make use of it to set flags for jumps which can then make use of the unchanged byte in the accumulator.

Finally, we have the test and jump set of instructions. There are two groups, the absolute jumps and the relative jumps: the absolute jumps are written in assembly language as JP and the relative jumps as JR. Each of these can be unconditional, meaning that the jump will take place no matter what flags are set or reset in the status register. Alternatively, they can be conditional, meaning that the jump will take place only if some selected flag in the status register is set or reset according to the programmer's choice. It's rather like the difference between GOTO 300 and IF A = 0 THEN GOTO 300. When we write conditional jumps in assembly language, we have to specify what the condition is, because each different condition means a different operating code, for example:

JR Z,disp

meaning jump to the new address if the zero flag is set, or:

JR NZ,disp

meaning jump to the new address if the zero flag is NOT set.

The complete list of available jumps is illustrated in Fig. 33.4. Some of them are very seldom used in the type of programs that you are likely to want to write for the Spectrum. The JP instructions need to be followed by a complete two-byte address, the JR instruction by a single-byte displacement number.

| | |
|-----------|--|
| JP addr | Jump to the address given. |
| JP c,addr | Jump to the address given if the condition c is met. |
| JR d | Jump relative to PC address, using displacement d. |
| JR c, d | Jump relative to PC address, using displacement d if condition c is met. |

Conditions for JP

NZ - not zero
Z - zero
NC - no carry
C - carry set
PO - parity odd
PE - parity even
P - sign positive
M - sign negative

Note: There is a JP version, JP (HL) which takes a jump to the address held in the HL registers.

Fig. 33.4. The jump command, both absolute and relative, unconditional and conditional.

Interacting with Spectrum

The time has come to start some practical machine code programming with your Spectrum. This is not simply a matter of typing the assembly language lines, because unless you have loaded an assembler program, the Spectrum will simply ignore these commands. What you have to do is to find the machine code bytes that correspond to the assembly language instructions, POKE them into the memory of the Spectrum, place the address of the first byte into the PC register of the Z-80A in your Spectrum, and watch it all happen. It sounds simple, but there is quite a lot to think about and a number of precautions to take. To start with, Spectrum uses quite a lot of the RAM, as we have seen in the first few chapters of this section, for its own 'housekeeping' operations. If we simply POKE a number of bytes into memory without taking a few precautions, the chances are that they will either replace bytes that the Spectrum needs to use, or they will be replaced by bytes that the Spectrum places in these memory addresses as it forms its variable list table and all the other items that are put into memory when a BASIC program is run.

We can prepare a section of memory for machine code in two ways. One is to type a REM statement as the first line of a BASIC program, and fill it with spaces – as many spaces as there will be bytes of the machine code program. This was the method used for the ZX81, which had no other provision for machine code. The Spectrum allows a memory space to be reserved by using the BASIC word CLEAR, however. If you are using a 16K Spectrum, in which the last usable

RAM address is 32767 (denary), then the RAM from 32600 to 32767 is normally reserved for user-defined graphics. You can reserve more space for your own machine code bytes just under this area by using CLEAR. CLEAR 32500, for example, will cause the RAM between 32500 and 32600 to be reserved for your machine code programs, and if you do not use the defined graphics in your program, there is nothing to prevent you from using the space between 32600 and 32767 as well.

The next problem is how to place the address of the start of your machine code program into the PC of the Z-80A. Once again, there is a BASIC instruction which copes with this, USR. When USR is followed by an address, the computer places that address into the PC of the Z-80A (and into the BC register pair as well), so that your program will run. By way of a bonus, when the Spectrum returns to normal operation, it will make a number available to you. This number will be the number stored in the BC registers, and you will see it printed on the screen if you started your program by:

PRINT USR address

or you can assign it to a variable by using:

LET x = USR address

or ensure that nothing is printed or assigned by using:

RANDOMIZE USR address

Note that USR cannot be used by itself – it has to be used after a 'do-something' statement. You may not wish or need to have a number returned to BASIC, but it's very useful. If your program does not make any use of the BC registers, then what you get back is the address number that followed USR when the program was called.

The next thing is to ensure that the machine code program will stop in an orderly way. Nothing you have done so far will indicate to the Spectrum where the end of the machine code occurs, so that it will continue to read bytes after the end of your program until it encounters some byte that will cause a 'crash'. You can prevent this by making the last byte of each program a 'return-from-subroutine' byte, code 201 denary, C9H, which will automatically cause a return to BASIC when the program has been started by a USR address command.

There is one additional point that we don't have to worry about at the moment. When you use a machine code program, you are running on the same Z-80A microprocessor as is used for all the actions of the Spectrum. If we make use of the Z-80A registers, we must be quite certain that we are not destroying information that is needed for Spectrum. For example, if, at the instant that your machine code program started, the Z-80A contained in its BC registers the address of the start of the table of reserved words (looking for USR, perhaps), then it wouldn't be a good idea to replace this with something else. When a machine code program is called into action by using the USR instruction, however, this problem is taken care of for you. The contents of most of the registers are placed in a part of

reserved RAM which is often called 'the stack'. This, incidentally, is another reason for being careful as to how you place your machine code in the memory – if you wipe out any of the stack the Spectrum will quite certainly not like it. When the RET instructions are carried out, the register values are restored equally automatically, and normal BASIC action can resume. If you call a machine code program into action by any other method, as is possible using some assembler programs, then you will have to do this salvage operation yourself. The assembler language mnemonic for saving register contents is PUSH, and the registers are pushed in pairs, AF,BC,DE,HL, and so on in 16-bit sets. To get them back, POP instructions for the same registers are used, and the registers have to be restored in the correct order – last in, first out, like NEXTs after FORs. If you have used:

```
PUSH AF
PUSH BC
```

at the start of your program, then you need:

```
POP BC
POP AF
```

at the end. If you use:

```
POP AF
POP BC
```

then you will have interchanged the contents of AF and BC! This is sometimes done deliberately (it's one way of changing the status register contents, for example), but it's not a technique for the beginner.

For the moment, then, we'll forget about PUSH and POP because the problem simply doesn't arise when we use USR as a way of starting the program. There is one exception, however. Spectrum makes a lot of use of the IX and IY registers, and these do not appear to be saved when a USR instruction is carried out. It may be that they can be used safely if they are PUSHed before using and POPped after, but until you have had some experience in the use of Spectrum with machine codes, it's better not to make use of them.

Practical programs

With these preliminaries out of the way, we can start on some programs which are very simple, but which are intended to familiarise you with the way in which programs are placed in the memory of the Spectrum, and with the use of assembly language and machine code.

We'll take the simplest possible example first – a program which places a byte into memory. In assembler language, it reads:

```
ORG 32500 ;start address
LD A,85 ;load 85 immediate
LD (32510),A ;put into 32510
RET ;return to BASIC
```

The first line contains a mnemonic, ORG, which you haven't seen before and which isn't actually part of the Z-80 set. It's short for ORIGIN, and it's a reminder to you that this is the address of the first byte of the program, the first byte of reserved memory. We have chosen an address here which reserves much more room than we need, but for the sake of an illustration it's as good as any other, and it leaves plenty of room for longer programs. When you program with an assembler, this statement can be typed in (see Chapter 36), and the assembler will then automatically start allocating address numbers or even putting the bytes of code into memory at this specified address. Some assemblers do this indirectly, creating the machine code on tape or disk rather than directly into memory.

The next step in the programming is to write down the codes. The codes must be looked up, taking care to select the correct mnemonics. The code for LD A,N, which is the way in which the immediate load is written in lists, is 3EH or 62 denary, so this is the first byte of the program, because there is no code corresponding to the ORG 32500 statement. We can start a table of address and byte numbers with this entry:

32500 62

and then move on. The LD A,N command has to be followed by the other operand byte, the byte that you want to place in the A register, which is 85 denary. Therefore 85 is the byte that has to be stored in the next address, making the table look like:

32500 62
 32501 85

The next byte we need is the instruction code for LD(Addr),A, and this is 50 denary, 32H. It goes down into the table, and then it has to be followed by the address that we want to use, which is 32510. The snag here is that this address has to be converted into two-byte form, and the bytes written in the correct order, lower byte first. You'll find it simpler if you have a calculator handy, or Spectrum switched on! Using a calculator, find first $32510 \div 256$ – this gives 126.99218. Write down the 126, which is the higher byte number, and label it HB. Now carry out the calculator steps (the circles round the symbols indicate that each of them is on a calculator key):

126 \times 256 = \div + 32510 =

What you are doing is multiplying 126 by 256 to get 32256, and then subtracting this from 32510 to get 254, which is the lower byte. You can now write the address 32510 in low, high order as 254,126 into your table, which now appears as:

32500 62
 32501 85
 32502 50
 32503 254
 32504 126

There's one last entry at address 32505, the return byte 201. If you are unhappy about the way of finding the bytes corresponding to the address number, use the Spectrum program shown in Fig. 33.5.

```

10 PRINT "Please type address - use 0 to""stop":
INPUT d
20 IF d = 0 THEN GOTO 9999
30 IF d > 65535 THEN PRINT "Too large - exceeds
65535": PAUSE 100 : GOTO 10
40 LET msb = INT (d/256) : LET lsb = INT (d -
256 * msb)
50 PRINT "Address bytes are "; FLASH 1; lsb;
", "; msb
60 PRINT ' : GOTO 10

```

Fig. 33.5. A BASIC program for calculating the two bytes of an address number.

The next step is to place this short program into the memory. We have to start by clearing a space, using CLEAR 32500, and then POKEing the bytes in one by one. Since Spectrum, unlike the ZX81, can use READ ... DATA, that POKE operation is most simply done using a loop, and since we know how many bytes we have, we can use the loop to read the bytes and place them into the correct address, starting with 32500. The program reads:

```

10 CLEAR 32500
20 FOR n=0 TO 5: READ b
30 POKE 32500 + n,b
40 NEXT n
100 DATA 62,85,50,254,126,201

```

Note that we've used n=0 TO 5, rather than 1 TO 6, so that we start at 32500 rather than 32501. When you count the bytes, start your count 0,1,2 rather than 1,2,3... and all will come right.

When you RUN this BASIC program, nothing appears to happen. All that it has done is to place these bytes into memory, and it certainly has *not* caused the machine code program to run. Before we do this, we need some way of checking that the machine code actually does something! Type PRINT PEEK 32510, and ENTER, and you should see the result 0 appear, unless for some reason something has been stored at 32510 by an earlier program. If you have just switched on, the result should certainly be zero.

Now type PRINT USR 32500 and ENTER. The number that appears now is 32500. It's just an echo of what you requested, because the machine code program does not use the BC registers, and these are loaded with the USR address number before your program starts running, as we commented earlier. The Spectrum has, however, carried out its action. If you now type: PRINT PEEK 32510, you will find the number 85 printed. This has been placed into the memory at address 32510 by your short section of machine code program.

It's no more than the command POKE 32510,85 would do from BASIC, but it's a start, and the main thing is to get used to how machine code programs are written, placed into memory and run. If you now press NEW and ENTER, you will see the screen clear after the black rectangle appears, but your machine code program is NOT cleared, though the BASIC program that put it into place will have gone. If you type PRINT PEEK 32510 you will see that the result is still 85, and if you clear this piece of memory by typing:

POKE 32510,0

followed by ENTER, of course, then you can check by another PEEK that this memory has been cleared. Your program still exists, however, and can be used again to load the number 85 into the address 32510. This time, try:

LET x = USR 32500 (and ENTER).

Nothing appears on the screen as a result of this form of command (though variable x is allocated to 32500, and PRINT x will reveal this), and you will find when you use PEEK 32510 that the value stored here is 85 once again. If you want to clear the memory completely, you can type

CLEAR 32767 (for a 16K machine)

and ENTER, then NEW (ENTER). This will remove everything, and if you forgetfully type PRINT USR 32500 and ENTER now, some number may be printed on the screen, but the computer will 'lock-up', no key having any effect, or possibly go into its NEW routine. If you get a lock-up, which is very common when a machine code program goes wrong, then all you have to do is to switch off and on again. Your program will, of course, be lost when you do this, but see Chapter 35 for how to save machine code on tape.

We very seldom need to clear the memory out in this drastic way. If you have reserved space for a machine code program by typing CLEAR 32500, then you can write as many programs as you like in this space. If you use the same addresses again, then the most recent program will replace any previous ones, but as long as each program uses a 201 code to return to BASIC, the old program bytes that are still stored at other locations will not affect the new program. One thing to watch, however, is how you use memory for storage, thinking back to the way in which we stored the byte 85 in address 32510. If the address that you use for storage in this way is inside the range of addresses that you use for your program, then you will have to write your BASIC program so that some bytes are POKEd before this address and some after but none in. Don't assume that because you didn't place anything there, there is nothing in the address! You can make your BASIC program POKE a zero into the space if you want, but you must not place any of the machine code instructions in that address. You will have to be equally careful that you don't let the MPU read this piece of memory as if it contained an instruction, so it will have to be jumped around. Fig. 33.6 shows how this can be done, but a much safer way when you are starting with machine code is to make all your use of memory for storage of quantities like this in addresses that are not

```

LD A, (HL)      ; load accumulator
JR 1             ; jump over byte
(data byte)      ; data byte used in program
addr: LD C, A    ; another load
LD (ADDR), C    ; put into data

```

Fig. 33.6. Jumping over a data byte. This must be done if a data byte is stored within a program, because it must not be read by the MPU as if it were an instruction.

within the range of addresses used by the program. Any address which is beyond the RET or a final JP or JR back will be suitable. If necessary, don't fill in details of address numbers until you can be sure how long the program is going to be, or use addresses that you can be sure will be well clear of the program addresses, like 32599 for a program that uses only 32500 to 32550.

Now try another example. We'll be more bold this time, and come back with something in the BC registers. This, remember, is simply a convenience of the Spectrum operating system. It's not a feature of the Z-80A, but since we're dealing with the machine code programming of the Spectrum it seems daft not to make use of the features that it provides. The assembly language version is shown in Fig. 33.7. There are no addresses shown this time, but note that the RL A command

```

LD A, 85          ; 85 into accumulator
RL A              ; rotate left
LD B, 0            ; zero B
LD C, A            ; load in result
RET               ; return

```

Fig. 33.7. An assembly language program which will return with a byte in the BC registers.

(rotate left accumulator) consists of two bytes, 203 and 23, so that your BASIC program takes the form:

```

10 CLEAR 32500
20 FOR n=0 TO 7: READ b
30 POKE 32500 + n,b
40 NEXT n
100 DATA 62,85,203,23,6,0,79,201

```

As before, when this is RUN, nothing noticeable happens because it simply places bytes into memory. When you use:

```
PRINT USR 32500
```

and ENTER, however, the number 170 appears on the screen, and this demands some explanation. What the program has done is to load the number 85, which in binary is 01010101 into the accumulator. A left rotation of this, Fig. 33.8, gives the number 10101010 in binary, and this in denary is 170. By loading register B with

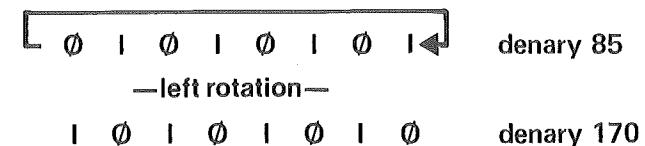


Fig. 33.8. Explaining the number which appears.

zero, and carrying out LD C,A so that the single-byte number 170 is copied from A to C, you end up with 170 in the BC register when the program returns. As Spectrum will always return with the number in the BC register pair (unless RANDOMIZE 32500 is used), the PRINT part of the USR call will ensure that this number is printed. Had you used LET x = USR 32500, then the value allocated to x would have been 170.

Now try this. List your BASIC program, and delete the numbers 6,0 from the DATA list, being careful not to leave a space with two lots of commas. The DATA list will now read:

```
100 DATA 62,85,203,23,79,201
```

and you will have to alter line 20 to read FOR n=0 TO 5 in place of the old count of 7. Now RUN to place the codes into memory, and use PRINT USR 32500 again. This time you get the number 32426 appearing. Why?

The answer is simple. The Spectrum operating system places the address that follows USR into the BC register pair. The address 32500 has a high byte of 126 and a low byte of 244 (so that $32500 = 126 \times 256 + 244$), so that when USR 32500 is called up, the content of the B register is 126, and the content of the C register is 244. The revised program has omitted the LD B,0 step, so that register B will still have 126 in it when the program ends with register C holding 170. This gives the denary number $126 \times 256 + 170$, which is 32426, in the BC register when the program returns. Automatic procedures like this can be very useful, but you have to keep your wits about you, because unless you do something to change numbers, they stay stored!

Chapter Thirty-four

Programming in Machine Code

The simple programs that we looked at in Chapter 33 don't do much, though they form very useful practice in converting assembly language into code bytes, putting them into the machine and running the resulting program. In this chapter, we shall look at how simple machine code programs are designed – in other words, how to get to the assembly language version, because this is by far the most difficult part of the process for the beginner to machine code programming.

The difficulty, curiously enough, doesn't arise because machine code is difficult but because it is simple. Because machine code is so very simple, you need to use a large number of instruction steps to achieve anything useful, and when a program contains a large number of steps, it's more difficult to plan. The most difficult part of the planning is breaking down what you want to do into a set of steps that can be tackled by assembly language instructions, and for this part of the planning, flowcharts are by far the most useful method of finding your way around. I never feel that flowcharts are ideally suited for planning BASIC programs, but for machine code, they can be very useful indeed.

Flowcharts

Flowcharts are to programs as block diagrams are to hardware – they show what is being done (or attempted!) without going into any more detail than is needed. A

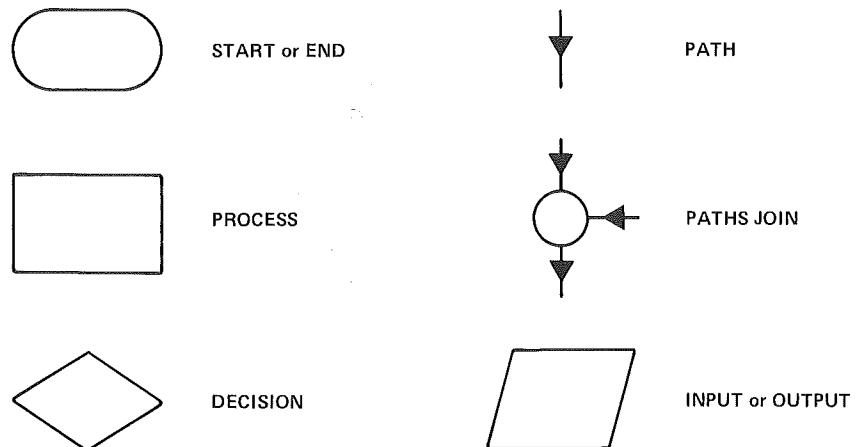


Fig. 34.1. Flowchart symbols – a small selection from the BS range.

flowchart consists of a set of shapes, each one meaning some type of action. Fig. 34.1 shows some of the most important flowchart shapes for our purposes (taken from the British Standard set), the terminator (start or stop), input/output, process (action) and decision steps. Inside these shapes we write the action that we want, but without details.

An example is always the best way of showing how a flowchart is used. Suppose you want a machine code program that takes the ASCII code for a key that has been pressed, and prints the character corresponding to that code on the screen. A flowchart for this action is shown in Fig. 34.2. The first 'terminator' is START,

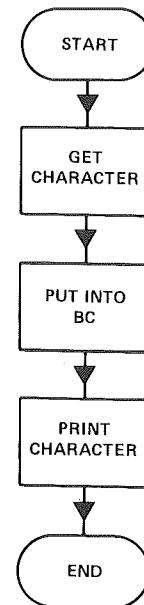


Fig. 34.2. A flowchart for the 'get character' program.

because every program has to start somewhere, and this leads to the first 'action' block, which is labelled 'get character'. This describes what we want to do – how we do it is something we don't know yet. After getting the character, the next 'action' block is: 'put into BC', because that is what we need if the value of the ASCII code is to be returned to the BASIC routine. Following that, the 'print character' block is something we can do in BASIC (it's not so straightforward in machine code). The END terminator then reminds us that the program ends here – it's not an endless loop.

This is a very simple flowchart, with no decision steps or loops, but it is enough to illustrate what we mean. Note that the descriptions are fairly general ones, so don't ever put assembly language lines into the action boxes of a flowchart, for example, because that is just downright confusing. Strictly speaking, I shouldn't have used the 'put into BC' box, but this is so essential to getting a number back into BASIC that it really needs to be mentioned as a reminder. A flowchart should

show anyone who looks at it what is going on; it shouldn't be something that only the designer can understand and which serves mainly to confuse everyone else. A lot of flowcharts, alas, are constructed after the program has been written in the hope that they will serve to make the action of the program clear. You wouldn't do that, would you?

Once we have a flowchart, we can check that it will actually do what we want by going over it very carefully. In the example of Fig. 34.2, the parts labelled 'get character' and 'put into BC' are going to be done using machine code, so we'll concentrate on them for now.

Getting the ASCII code for a character looks tricky at first. A lot of computers put the ASCII code into the accumulator during the subroutines of reading the keyboard, and then store the value temporarily in RAM. Looking at the description of the Spectrum system variables in Chapter 25 of the manual reveals that address 2356 \emptyset is used for this purpose – the code for the last key pressed is stored here. If we load A from this address, we should then have in the accumulator the ASCII code for the last key that was pressed; step 1 completed. The next step is already familiar – we want \emptyset in the B register, and to copy the byte in the A register into the C register. This is necessary because we can't load the C register directly from a memory address – that's one of the restrictions which makes the A register more useful for a lot of purposes than any other single-byte register. Having loaded BC, we can then return to BASIC, and make use of the code in BC in our BASIC program. If we call the program by using LET x = USR 3250 \emptyset , then we can print CHR\$ x to display the character.

That's half of the problem overcome. The next part is how to get the byte into address 2356 \emptyset using BASIC. If we use INPUT for this stage, then the byte in 2356 \emptyset will always be 13 – because that's the code for the ENTER key that you have to press to enter your input! Perhaps INKEY\$ will be more useful – we can set up a loop so that pressing a key will leave the loop and call the machine code program when the value of that key is in address 2356 \emptyset .

We can start by designing the assembly language program, which might read:

```

LD B,0      ;clear B
LD A,(23560) ;get ASCII code
LD C,A      ;put into C
RET         ;back to BASIC
  
```

which looks as if it should do what is wanted. To call it and use it, we can have:

```

200 LET k$ = INKEY$ : IF k$ = "" THEN GOTO 200
210 LET x = USR 32500
220 PRINT CHR$ x
  
```

Now we can complete the operation by looking up the codes and writing the part of the BASIC program that will POKE them into memory:

```

10 CLEAR 32500
20 FOR n = 0 TO 6 : READ b
  
```

```

30 POKE 32500 + n, b
40 NEXT n
100 DATA 6,0,58,8,92,79,201

```

Add this to your lines 200 to 220 and we should be all set.

Sure enough, when this runs, we can press a key and we will find the character printed on the screen. Once again, it's not terribly impressive because it's no more than we would get if we programmed in BASIC:

```

200 LET k$ = INKEY$ : IF k$ = "" THEN GOTO 200
210 PRINT k$

```

The aim, however, is to get experience of how machine code programs can be written and can interact with the Spectrum BASIC, not to start writing original works of genius.

Now let's take a look at some ways of cutting out some flab from this program. To start with, we find that we don't need two lines for 210, 220 because we can substitute the single line:

```
PRINT CHR$ USR 32500
```

If it looks odd, think of it as PRINT the character whose code is obtained from the USR 32500 subroutine. Now the next bit we have to tackle is the INKEY\$. Do we need to use BASIC here? The answer is that we can create our own INKEY\$ loop, using the program shown in Fig. 34.3. The machine code section simply places the

```

LD A, (23560)      ; get last character
LD B,0              ; clear B
LD C, A             ; load in result
RET                ; back to BASIC

10 CLEAR 32500
20 FOR n = 0 TO 6; READ b
30 POKE 32500 + n, b: NEXT n
50 DATA 58, 8, 92, 6, 0, 79, 201
100 LET x = USR 32500: IF x <= 32 THEN GO TO 100
110 PRINT CHR$ x

```

Fig. 34.3. A get-character program. The BASIC part is needed to place the character into memory.

byte at address 23560 into the BC register as before, but the BASIC section then tests this, and if the byte is less than 32 (the space byte), then the loop repeats until a byte greater than 32 is found. This doesn't completely release us from BASIC, because we are still using line 100 to make the test. Could we test the byte and loop back within the machine code program? The answer for the moment is no, because to get a new byte into 23560, the Spectrum operating system must be running, and it can't run while our own program is continually looping round to test 23560 in

machine code! If we want to test key values, then we have to use other ways, as we shall see later.

Let's look at another aspect of input for the moment. Suppose we wanted automatic upper case (capital letter) shift, so that when we pressed the h key we would get H printed. Looking at the ASCII codes reveals that the lower case letter codes use numbers which are 32 greater than the corresponding upper case codes, so we might try subtracting 32 from these lower case codes to get the upper case letters. What we need to be careful about though is what happens if we already have typed an upper case letter. We don't want this change to be made when the ASCII code is 96 or less, because this is the code for a (A is 97 - 32 = 65).

A flowchart for a simple scheme is shown in Fig. 34.4. We want to get the code, and compare it with 97. If it is equal to 97 or greater, we shall subtract 32, but if the code is less than 97, we shall leave it unchanged. Either way, we then load the code into BC and that's the end of the machine code section.

Now this introduces a decision step which we haven't used in assembly language before, so we'll take the next few stages slowly. Fig. 34.5 shows the assembly language version of the flowchart. The character code is obtained as before by

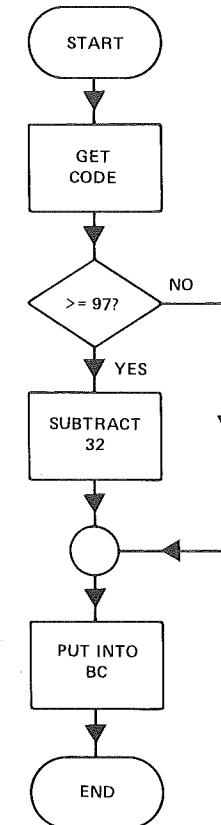


Fig. 34.4. A flowchart for the conversion to upper case program.

```

LD A, (2356$)      ; get character
CP 97              ; compare
JR C, Out          ; out if less than 97
SUB 32              ; otherwise subtract 32
Out: LD B, $        ; zero B
LD C, A            ; transfer answer
RET                ; back

```

Fig. 34.5. The assembly language version of the conversion program.

loading the accumulator from address 2356\$, but the decision step needs two instructions. The first one is CP 97, meaning compare whatever number is in the accumulator with 97. As we saw earlier, it's like subtraction, but without the content of the accumulator being affected. If the number in the accumulator is 97 or greater, then it will be possible to subtract 32 from it, but if the byte in the accumulator is less than 97, the subtraction will require a 'borrow', which is indicated by the carry flag of the status register being set to 1. It will be \$0 if the byte in the accumulator is 97 or greater. If the carry flag is set by the CP step, then, we do not need to subtract 32 from the code number. If the carry flag is not set, then we do have to subtract 32. This part is done by a decision step, a jump. In the assembly language version, the destination of the jump is indicated by using a word, Out. This is a label word, which is a convenient way of indicating on an assembly language program where a jump is to go, because we have no addresses written down, and it would be awkward to calculate displacement bytes at this stage. The destination is confirmed by having the same word placed in front of the step to which the jump must go, the LD B,\$0 step. If the number in the accumulator is 97 or more, however, the carry flag will not be set, the jump will not occur, and we need to subtract 32 from the byte in the accumulator. This is done by the next step in line, SUB 32, which is then followed by the LD B,\$0 command. A close scrutiny of the assembly language program shows that it should do what we want it to do; it certainly follows the actions of the flowchart.

The next step is coding. This is straightforward enough apart from the jump step. The difference between the JR address and the LD B,\$0 address is 4 bytes, so we should get the correct displacement byte, following the rule given in Chapter 33, by subtracting 2 to get a displacement of 2. The result is the program shown in Fig. 34.6.

There is an unexpected line 13\$ in this program. The program will work perfectly if the lines after 12\$ are omitted, but will print just one letter on each run when a key is pressed. It is much more useful when it is used in a loop, but if you simply add GOTO 10\$, you will find curious effects, letters repeating, or unwanted spaces. This is because the buffer memory which is used for the INKEY\$ function is not always cleared in time, and line 13\$ ensures that it is by holding a loop until INKEY\$ is a blank. You'll find that this program gives you an upper case letter each time you press a letter key.

Once again, it's not a masterpiece (what happens when you press a number

```

1$  CLEAR 325$0
2$  FOR n = $0 TO 12
3$  READ b: POKE 325$0 + n, b
4$  NEXT n
5$  DATA 58, 8, 92, 254, 97, 56, 2, 214, 32,
   6, $0, 79, 20$0
10$ LET k$ = INKEY$: IF k$ = "" OR CODE k$ =
   13 THEN GOTO 1$0
11$ LET x = USR 325$0
12$ PRINT CHR$ x;
13$ IF INKEY$ <> "" THEN GOTO 13$
14$ GOTO 1$0

```

Fig. 34.6. The coded version of the conversion program.

key?). It could have been done completely in BASIC by using some of the reserved RAM addresses – if you try the program of Fig. 34.7, then you will see that most of the keys will give the correct upper case characters though others give the query sign (which means that the number stored at that address has no ASCII code). Once again, what we are doing is something that can be done with BASIC, but the aim is to learn machine code methods, and if we stuck to examples of things which could not be done in BASIC, we would end up with machine code examples which would be just too difficult to follow at this stage.

```

1$  LET k$ = INKEY$: IF k$ = "" OR CODE k$ =
   13 THEN GOTO 1$0
2$  LET x = CODE k$
3$  IF x >= 97 THEN LET x = x - 32
4$  PRINT CHR$ x;
5$  IF INKEY$ <> "" THEN GOTO 5$0
6$  GOTO 1$0

```

Fig. 34.7. The conversion program in BASIC.

In any case, we have now used a decision step in a machine code program as well as in flowchart and assembly language form. It's another step on the way for us, and some more practice in loading and running machine code programs. Perhaps we can now try something more ambitious, and at the same time probe the secrets of the Spectrum rather more deeply. Earlier on, we found that there were certain difficulties attached to trying to simulate the action of INKEY\$ without using BASIC. Well, now is the time to show the way. There is a machine code routine in the ROM for reading the keyboard, and if we can find it, then we may be able to make use of it in our programs. Finding the routine with the aid of the Campbell disassembler wasn't difficult. I looked for a piece of program that loaded the address 2356\$, and having found this, traced it back to find where it started. This led me to the address \$02BFH, 703 in denary, as the start of a routine which read

the electrical signals from the keyboard and converted them into ASCII codes. A preliminary try-out showed me that the routine placed 255 in the accumulator if no key was pressed, and the ASCII code for the key if a key was pressed.

Now the new feature for you at this stage is how you can make use of a Spectrum ROM routine. A subroutine in BASIC is called by using GOSUB, and the return is forced by using the BASIC command RETURN at the end of the subroutine. In assembly language, a subroutine is called by CALL, followed by the address of the start of the routine. The return is forced by the RET instruction which we have already used. CALL and RET have to be used together, and the reason that we were able to use RET at the end of our routines to return to BASIC is that the CALL part of the process has been done by the USR routine.

We can get a byte into the accumulator from the keyboard by using CALL 703 as an instruction. The CALL code is 205, and the address which is called must be written in the usual low-byte, high-byte form (you must follow CALL by two bytes of address, even if one is zero). The address 703 is coded as 191,2, so that the set of numbers 205,191,2 will carry out the action of CALL 703.

Fig. 34.8 illustrates the flowchart we shall need for this program. The byte is fetched from the keyboard, using the CALL, and tested for equality to 255, because this is the number that the ROM routine will put into the accumulator if no key is pressed (it's the number that means 'false'). If the byte equals 255, then the call is repeated, but if a key has been pressed, then the value in the accumulator will

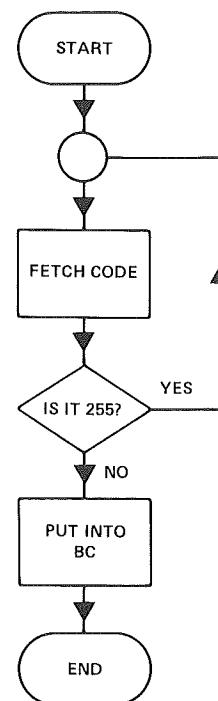


Fig. 34.8. The flowchart for a key-reading program.

be the ASCII code for that key. This value in the accumulator can then be passed to the BC registers as before.

Fig. 34.9 shows a BASIC program that POKEs the code into memory and uses it. Just for a change, I've illustrated a quicker way of using the POKE and USR commands. By having LET j = 32500 early in the program, we can save memory by using j in place of 32500. Since this saves a conversion from ASCII to binary each time 32500 is used, it saves time. The program does in machine code what INKEY\$ does in BASIC; it waits for a key to be pressed, and then returns with the value of the ASCII code in the accumulator.

```

100 CLEAR 32500
200 LET j = 32500
300 FOR n = 0 TO 10
400 READ b: POKE j + n, b: NEXT n
1000 DATA 205, 191, 2, 254, 255, 40, 249, 6, 0,
    79, 201
2000 PRINT CHR$ USR j
  
```

| | |
|--------------------|------------|
| Assembly language: | Back: |
| | CALL 02BFH |
| | CP 255 |
| | JR Z, Back |
| | LD B, 0 |
| | LD C, A |
| | RET |

Fig. 34.9. The BASIC program which POKEs the code and makes use of the machine code routine.

This is the first use that we have made of a routine in the ROM, and it's not always something that we can do easily. It's not necessarily easy to find ROM routines unless you have a disassembler, lots of time, and some confidence in reading assembly language code - it was fortunate that the INKEY\$ routine was so easy to spot. There are, however, complete commented disassemblies of the ZX81 ROM available, so it's just a matter of time before we find some similar treatment for Spectrum (some of the routines are almost identical, though not always in identical addresses). When this is done, you will be able to look up the routines and use them for yourself, though you must be careful to make sure that your registers are loaded correctly before calling the routines.

It's time now to try something for yourself. Can you combine the routine we have just examined, the machine code equivalent of INKEY\$, along with the idea of subtracting 32 if the byte is 97 or more, to get a routine which will put all printing into upper case?

Loops

The loop action in BASIC that you use most of all on the Spectrum is the FOR ... NEXT loop. This uses a 'counter' variable to keep a score of how many times you

have used the loop, and compares the value of the counter with the limits you have set on each pass through the loop. Now the action of a FOR ... NEXT loop can be simulated without using FOR ... NEXT, as is shown in Fig. 34.10, using an IF ... THEN line to make the decision as to whether or not to keep looping back. Most types of microprocessors can tackle looping in a similar way, using one register or a memory address to keep a count of what is going on. The Z-80 family, however,

```

10 LET count = 0: LET end = 10
20 PRINT "Action "; count
30 LET count = count + 1
40 IF count <= end THEN GOTO 20
50 PRINT "Finished"

```

Fig. 34.10. How the action of a FOR..NEXT loop can be simulated in BASIC.

has a set of ready-made loop instructions, and these can save a considerable amount of programming.

The Z-80 equivalent of FOR ... NEXT is written in assembly language as DJNZ, short for decrement and jump if not zero. Register B is used as a counter, and each time the DJNZ code (10H = 16 denary) is encountered, the number in the B register is decremented and the register is tested, using the flags in the status register, to see if the decrement action has resulted in the number reaching zero. If it has not been decremented to zero, then a loop back is taken, using a displacement byte in the same way as a JR instruction. If the register content has reached zero, the instruction following the DJNZ one is carried out.

Fig. 34.11 illustrates this in a program which simply loads B with 255, the maximum possible size of a single byte (DJNZ treats this as being *unsigned*). The byte which causes the jump back is -2 (254), so that the jump is simply to the start

```

Back: LD B, 255
      DJNZ Back
      LD C, B
      RET

```

```

10 CLEAR 32500: LET j = 32500
20 FOR n = 0 TO 5: READ b
30 POKE j + n, b: NEXT n
50 DATA 6, 255, 16, 254, 72, 201
100 INPUT "Press any key ..."; a$
110 PRINT USR j: PRINT "Done"
120 PRINT "Now try a BASIC count ..."
130 INPUT "Press any key ..."; a$
140 FOR q = 255 TO 0 STEP -1: NEXT q
150 PRINT "Done!"

```

Fig. 34.11. A countdown program in machine code, using DJNZ, with the BASIC version to compare speeds. The result is misleading!

| | |
|---------------|-------------|
| LD BC, FFFFH | 1, 255, 255 |
| Count: DEC BC | 11 |
| LD A, B | 120 |
| OR C | 177 |
| JR NZ, Count | 32, 251 |
| RET | 201 |

Fig. 34.12. Assembly language version of a countdown of a much larger number stored in a register pair. Note the method used to check the end of the count.

of the DJNZ instruction again. The program lets you see how fast this count is performed compared to a FOR ... NEXT loop using the same numbers. In fact, the machine code count is very much faster than it appears, because of the time that is needed for the BASIC part of the program to respond in lines 100 and 110 – you see it return by the 0 appearing on the screen because this is the number in BC.

The machine code count in this last example is so fast that we can't time it, and the time that is taken is practically all due to the BASIC part of the machine code program, so it's interesting to compare a much longer count. DJNZ can be used only for a single-byte count; there is no provision for a DJNZ count in a double register, so that this is a good opportunity to look at a method that is used for counts of more than one byte. We start by loading the BC register pair, in this example, with the largest number that we can get in two bytes, FFFFH, which is 65535 denary. Counting starts by decrementing BC, but because decrementing a pair of registers does not, on the Z-80, cause any flags to be affected, we have to test for the end of the count in a different way. The count will be complete when both registers of the BC pair have reached zero, and the standard method of checking this is to load one of the registers into A, and then OR it with the other one. If either of the registers contains a 1 bit anywhere in its byte, the OR will produce a 1 in the same place in the result, and the zero flag will not be set. As a result, the program loops back at the JR NZ stage - the jump is three address numbers back, which is -5 denary, 251 as a displacement byte. Fig. 34.12 shows the program and the denary numbers for the machine code.

Now if you substitute this for the previous steps, as shown in Fig. 34.13, you will find that the machine code count takes about half a second. The BASIC count of

```

10 CLEAR 32500: LET j = 32500
20 FOR n = 0 TO 8: READ b
30 POKE j + n, b: NEXT n
50 DATA 1, 255, 255, 11, 120, 177, 32, 251, 201
100 INPUT "Press any key ..."; a$
110 PRINT USR j: PRINT "Done"
120 PRINT "Now try a BASIC count ..."
130 INPUT "Press any key ..."; a$
140 FOR q = 65535 TO 0 STEP -1: NEXT q
150 PRINT "Done at last !"

```

Fig. 34.13. A program which allows you to compare counting speeds of BASIC and machine code over the same range of numbers.

the same number, however, takes several minutes, and this is a more accurate picture of how much faster machine code instructions can be. We can, incidentally, make use of this sort of loop for timing. If the crystal of the Spectrum is running at a speed of 3.5 MHz, meaning $3\frac{1}{2}$ million pulses per second, then by counting the number of pulses that are needed for a program we can find how long it takes. Fig. 34.14 shows how these times are applied to our loop. The sum of

| Operation | Time | Comment |
|--------------|------------------------|-----------------|
| DEC BC | 6 | Same as for INC |
| LD A, B | 4 | |
| OR C | 4 | As for ADD C |
| JR NZ | 12 | In each loop |
| TOTAL | 26 clock cycles | In loop |

Fig. 34.14. The times for the loop instructions added together.

pulses in the loop between DEC BC and JR NZ is 26 pulses, so that to go round the loop 65535 times will need about 1,703,910 pulses. We can add to this, if we like, the time needed for the LD BC and the RET instructions; strictly speaking we should count the time for JR NZ for only 65534 loops, and use a shorter time for the last run, which doesn't loop back. These, however, are small corrections to a large number. By this reckoning, we should have finished the loop in a time of $(1703910)/(3500000)$ seconds, about 0.48 seconds, which seems to correspond with experience.

Counting can produce time delays which, since the clock rate is controlled by a crystal of quartz (like digital watches), are very precise. These time delays are used to a considerable extent in the Spectrum ROM – for the TV display, the BEEP routine, cassette input and output, and the printer routines to name the most obvious examples.

Chapter Thirty-five

Using the Ports

Storing and reloading

Up to this point, we have generated machine code programs as BASIC routines which POKEd the numbers into memory. This is easy to carry out, and there is no reason why the whole program should not be saved as a BASIC program. If you use the Campbell disassembler, you will find that you can use it to place code in memory, using hex rather than denary, and this can be faster than writing READ ... DATA loops in BASIC. You then have the problem of how to save such a program on tape, when there is no BASIC program that has generated it. The same problem arises when you have used the ASC ULTRAVIOLET assembler to produce machine code which must be recorded and relocated. Even if you have used a BASIC program to place the bytes in memory, you may want to record the machine code separately so that you can place it into the Spectrum without having to load in the BASIC program that performs the POKE routines.

Happily Spectrum, unlike its predecessors, provides for saving and loading machine code directly though, as you might expect, you have to be much more specific about what you want. The syntax is summarised in Chapter 30 of the Spectrum manual, but some practice with one of our programs will give you more confidence in the use of this set of commands.

Let's use the program that we finished with in Chapter 34, which stores 9 bytes starting at address 32500. The syntax of a SAVE command for this code will be:

SAVE "Count" CODE 32500,9

You can choose your own name for the program of course. When you press ENTER you will get the usual message about starting the recorder and pressing any key. The program saves in much the same way as a BASIC program, so keep a note of where it is on the tape so that you can wind back to the start again.

To make a fair test, you will then have to switch off the computer (perhaps you might like to save the BASIC POKE program first, just in case!) so that the entire memory is cleared when you switch on again. Switching off and then on again re-allocates all of the memory, so before you try to load the program again, you will have to type CLEAR 32500 so as to reserve space for your program. Having done this, type:

LOAD "Count" CODE 32500,9

and when you press ENTER and start the recorder on PLAY, your code will be loaded in again, starting at address 32500. You will see the message:

Bytes: Count

to remind you that this is a machine code program called Count. When the bytes have completely loaded you will get the usual \emptyset OK, \emptyset :1 message at the foot of the screen when loading is complete.

There are a number of variations on the LOAD part of this set of commands. Using the full version above keeps a careful check for errors, so that if there are more bytes recorded on the tape than you provided for in the length number, you will get an error report. If you have forgotten or don't know how many bytes there are, you can use:

LOAD "Count" CODE 325 \emptyset

This will start the loading at address 325 \emptyset , and load in as many bytes as are recorded on the tape. If you run out of memory, too bad – you'll have to try loading at a different address. Not all programs will still work when they are loaded at a different address, but this particular one will, so with the program loaded at 325 \emptyset , try reloading with:

LOAD "Count" CODE 325 \emptyset 6

This will carry out the loading, starting this time at 325 \emptyset 6, and replacing any bytes that existed previously at these addresses – there will be some left over from the program that was loaded in at 325 \emptyset . You can now call this program up to check it by using:

PRINT USR 325 \emptyset 6

which will return with zero after a short delay.

What you must not on any account do now is to use PRINT USR 325 \emptyset again. You might get away with it (in this example you will because of the way the program happens to work), but generally the result will be a locked out computer, with no keys having any effect. Switch off and start again.

If you know neither the starting address nor the length of a machine code program on tape, then Spectrum can still cope. By using the form:

LOAD "Count" CODE

with no numbers, the Spectrum will load the bytes into the position in the memory that they occupied when they were recorded, which is 325 \emptyset onwards in our example. This is always a safe method provided that you have cleared enough memory space.

Coming back to the idea of loading machine code into a part of the memory which is not the same as was used originally, this is possible only if the machine code is *position independent*. Position independent code is code which uses no full addresses that are within the program or within the range of addresses to which the program will be shifted. For example, suppose you have a program which starts at 325 \emptyset and ends at 32599. If in the program there is an instruction such as JP 3251 \emptyset or CALL 32577, then these address numbers will be written into the program as code. If you then tried to load this program at addresses 320 \emptyset to 32099 (assuming

that you had cleared enough memory), you would find that it did not run and you would get a lock-up. Why? Because you still have instructions JP 3251 \emptyset or CALL 32577 which refer to addresses where by now there may be completely different codes, or only zeros stored. Code like this can't be relocated simply, and a similar thing applies if you had a call to 320 \emptyset in a program that was placed at 325 \emptyset ; if you relocated this program to 320 \emptyset , you would overwrite the section of code which you wanted to call. Relocating a program of this type is not possible unless all of these addresses are changed (see Chapter 36 for relocating programs created with the ULTRAVIOLET assembler). If, however, your program had all its jumps in the JR form and all calls to the operating system which, being in ROM, is always at a fixed address that cannot be overwritten, then you could place such code anywhere in the RAM and run it. This type of code is 'relocatable', and the LOAD "Name" CODE start address type of command can be used to place it wherever you want it, assuming that you have cleared memory for it.

There is, incidentally, a VERIFY routine for machine code so that you can check if a valuable program has recorded correctly. This is particularly useful if the code has not been generated from a BASIC program using a READ ... DATA and POKE routine.

Berthing at a port

A *port*, as we saw in Chapter 19, is a circuit used to send signals into or out from the microprocessor system of MPU, ROM and RAM. We also saw that as far as the MPU is concerned, a port is just another address which can be used rather like memory, but with a more restricted instruction set. The simplest Z-80 port instructions are those using the IN A,(port) and OUT (port),A assembly language instructions, and Spectrum BASIC provides commands which perform port action using the keywords IN and OUT. As you might expect, the machine code version is faster, and requires a much more detailed specification.

First, though, it's probably worth reminding yourself of the way the BASIC commands work by looking back to the end of Chapter 20 for a moment, where we discussed the way in which signals from the keyboard are read. Fig. 35.1 shows

| Code (Denary, Hex and Binary) | | | Key position in half-row |
|-------------------------------|----|-----------------------|--------------------------|
| 254 | FE | 1111111 \emptyset | 1st |
| 253 | FD | 1111111 \emptyset 1 | 2nd |
| 251 | FB | 111111 \emptyset 11 | 3rd |
| 247 | F7 | 1111 \emptyset 111 | 4th |
| 239 | EF | 111 \emptyset 1111 | 5th |

Note how much better the pattern is shown by the binary version of the code.

Fig. 35.1. The codes that are read in from the keys at each port.

the actual codes that are returned from each half-row when keys are pressed; note that these same numbers are read in even if the shift keys are pressed at the same time. The keys that are accessed by different ports cause the same sequence of numbers to be generated, so we can use a program like that of Fig. 35.2 to check for two keys being pressed. This is rather elementary BASIC, and we could program it more neatly – but no matter, it does what we want.

```

10 PRINT "Press g and t"
20 LET x = IN 65022: LET y = IN 64510
30 IF x = 255 OR y = 255 THEN GOTO 20
40 IF x = 239 AND y = 239 THEN PRINT "That's
it!": GOTO 9999
50 PRINT "Cheat!"
```

Fig. 35.2. A BASIC program to check for two keys being pressed at the same time.

How would this work in assembly language? We have to start by writing a flowchart, and Fig. 35.3 shows a suggestion. The problem that we are tackling is one of reading two ports, and only when both report a key-press do we proceed – this will be when neither port comes in with 255. In the flowchart, the first port is tested, and if it reads 255, the program loops back to test this port again. If the reading is less than 255, indicating that a key in this half-row is pressed, then the second port is tried. If this one reads 255, meaning that no key is pressed in the second half-row, the program loops back to the start again. If we only looped back to the second test, this would be testing for two keys pressed in sequence rather than at the same time – not what we want, though it can be useful (you could make a program do one thing by typing GO and another by typing NO, for example). When both keys are pressed in the different half-rows, we put the resulting numbers into B and C (one byte in each) so that they can be passed back to BASIC.

It all looks very reasonable, but if we use the commands IN A,(port), then the assembly language requires a port address of one byte only, and the addresses in the manual consist of two bytes. If we attempt to load the accumulator from these addresses, rather than from a port, we find that they do not exist on the 16K Spectrum – there is no memory there. The clue lies in the type of port input command we need to use. The Spectrum uses a different type of port command for its keyboard reading routines, one which makes use of a two-byte port address in the BC register pair rather than the single-byte method of IN A,(port). When the port input command IN A,(C) is used, the number that is held in the BC pair is used as an address, the lower half in the C register being used as the port address. At the same time, the half address on the B register is sent out on the address lines, and this is how the Spectrum addresses its keyboard ports. The same scheme was used by the ZX81 as you might expect.

To make our flowchart work, then, we must place suitable addresses in the BC register pair, and the addresses are just those shown as port addresses in the manual, though you will have to split them into the usual two-byte form to load

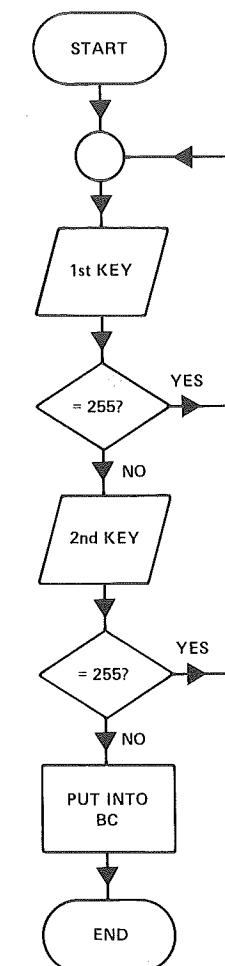


Fig. 35.3. A flowchart for an assembly language version of the two-key program.

| | | |
|-------|--------------|-----------|
| Strt: | LD BC, 65022 | 1,254,253 |
| | IN A, (C) | 237,120 |
| | CP 255 | 254,255 |
| | JR Z, Strt | 40,247 |
| | LD D, A | 87 |
| | LD B, 251 | 6,251 |
| | IN A, (C) | 237,120 |
| | CP 255 | 254,255 |
| | JR Z, Strt | 40,239 |
| | LD C, A | 79 |
| | LD B, D | 66 |
| | RET | 201 |

Fig. 35.4. An assembly language program for detecting two keys. This uses the IN A, (C) instruction.

them into BC. Fig. 35.4 shows the assembly language version of the two-key program, taking for the sake of example the ports that we used in the BASIC version. The ASDFG group of keys uses the address that codes as 254,253 (low, high order), and the QWERT group uses 254,251, so that if we load BC with 253,254 (high, low order) initially, we need only change B to detect the second half-row of keys. Note that we have to save the first value we find in register D, because A will be reloaded with that value, when the second key is detected. At the end of the program, the values are put into BC so that they can be returned for inspection. We would normally, of course, make some use of these values. Fig. 35.5 shows the numbers that are returned, which are identical to the numbers that would be returned by a BASIC version of the program.

```

100 CLEAR 32500: LET j = 32500
200 FOR n = 0 TO 24: READ b
300 POKE j + n, b: NEXT n
400 PRINT USR j
1000 DATA 1, 254, 253, 237, 120, 230, 31, 254, 31,
40, 245, 87, 6, 251, 237, 120, 230, 31, 254,
31, 40, 234, 79, 66, 201

```

| Keys pressed | Number | Analysis (B, C) |
|--------------|--------|-----------------|
| g and t | 3855 | 15, 15 |
| a and q | 7710 | 30, 30 |
| w and s | 7453 | 29, 29 |
| e and d | 6939 | 27, 27 |
| f and r | 5911 | 23, 23 |

The g and t example has been shown first because it was used previously.

Fig. 35.5. The BASIC POKE program for the assembly language program, and the results of pressing keys.

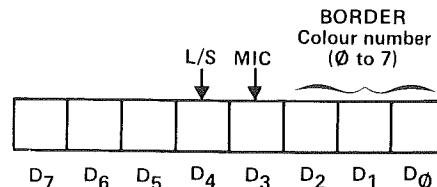


Fig. 35.6. How the Port 254 bits are allocated.

Let's try a port output now. Port 254 uses three bits to control the border colour, one for the MIC socket, and one for the loudspeaker. The byte is divided as shown in Fig. 35.6, with the lowest three bits (called D0, D1 and D2) forming the number for border colour, D3 sending out one bit at a time to the MIC socket, and D4 sending out one bit at a time to the loudspeaker.

We can try a short assembly language program to send something out to the

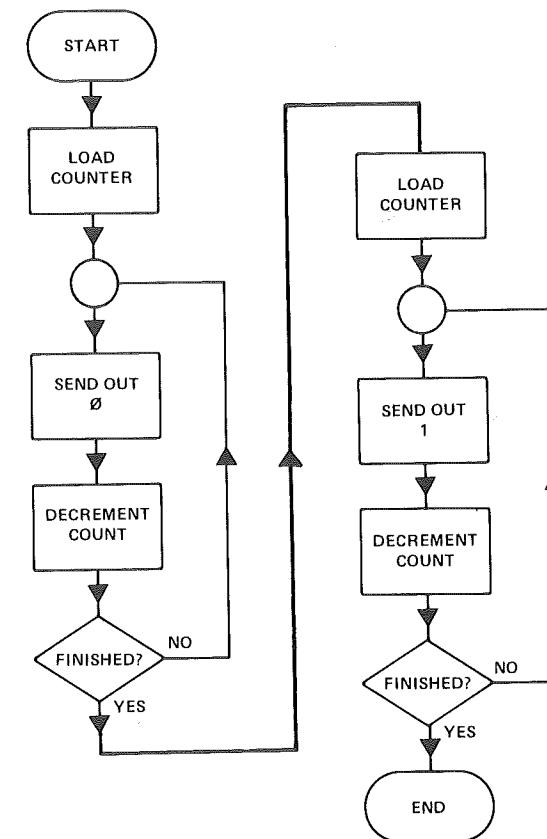


Fig. 35.7. A flowchart for sending bits out from the loudspeaker port.

loudspeaker bit of this port and observe the effect. The flowchart for this action is shown in Fig. 35.7, and the assembly language version with its numbers to POKE is shown in Fig. 35.8. When we put this into memory (Fig. 35.9) and run it, the most obvious thing that happens is that the border turns black! A little thought shows that this is logical, because we are using the numbers 0 and 16 to put out to the port in our program. 0 denary is 00000000 in binary, and 16 denary is 00010000 in binary. In each case, the three lowest bits are zero, corresponding to the number 0, if we think of these three bits only, and this presumably means border zero, which is black. Suppose we used the number 4 denary in these bits, which would mean 100 for the last three bits. This means loading the accumulator with 4 and 16 + 4 = 20 in place of 0 and 16; so we can alter the data bytes accordingly. Yes, this gives us a green border, so we've discovered how to control the border colour in machine code!

That's not what we were aiming at, however, and if you listen very carefully when you run the program each time, you will notice that when you press ENTER to run, you will hear a double click. One of these is the click you normally get when you press a key, but the other one is the result of your program. It has sent a 0 and then a 1 to the loudspeaker, causing one click.

```

LD B, 255      6,255
LD a, Ø        62,Ø
out1 :OUT (port),A 211,254
DJNZ, out1    16,252
LD B, 255      6,255
LD A, 16       62, 16
out2 :OUT (port),A 211,254
DJNZ, out2    16,252
RET            2Ø1

```

Fig. 35.8. The program in assembly language form.

```

1Ø CLEAR 325ØØ: LET j = 325ØØ
2Ø FOR n = Ø TO 16: READ b
3Ø POKE j + n, b: NEXT n
5Ø DATA 6, 255, 62, Ø, 211, 254, 16, 252, 6,
255, 62, 16, 211, 254, 16, 252, 2Ø1
1ØØ PRINT USR j

```

Fig. 35.9. The program in POKE form.

The next exercise is to prolong this into something more worth listening to. This means repeating the click often enough and fast enough to make a sound that will be more noticeable, and that in turn means another loop. Once again, we have to use a flowchart to see what we need to do, and Fig. 35.10 illustrates this one. It's a short flowchart, because the whole of the 'click' routine that we used in Fig. 35.7 has been shown as just one 'action' block rather than being drawn out in detail. The problem now is how to set about programming this action.

There are several ways, and it's instructive to look at more than one. Since we have another count to perform, it would be very useful if we could use DJNZ again. This, however, needs some care because we have used DJNZ twice in the click routine, and it always ends up with the B register counted down to zero. If we put a value into B to use as another counter, it will be overruled by the DJNZ commands in the click routines. What we can do, however, is to load B, PUSH this on to the stack, run the click routine, POP BC back, do the DJNZ and if the stored value has not decremented back to zero, jump back to the PUSH operation. Remember that the stack is a piece of RAM which is used by the microprocessor as a temporary store for register contents, so that the value we place there will be unchanged until we recover it and use it. If we carry out a PUSH BC before the click routine, then the number that was in the B register before the PUSH will be put back into the B register when we POP BC again.

The alternative method is to abandon the use of DJNZ for the number of click routines, and use a spare register or register pair for a count. A single register can be loaded with a value, and decremented after the click routine, then tested to make a jump back if it has reached zero. If a register pair is used, then remember that decrementing a register pair does not cause flags to be affected. Whichever

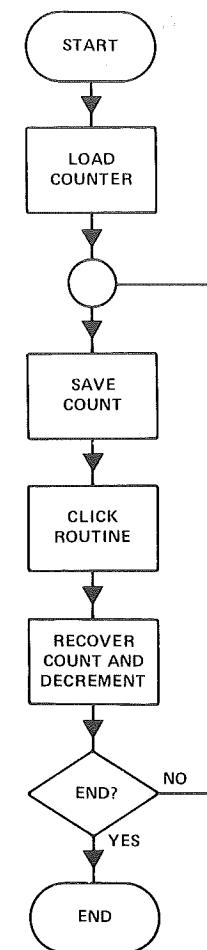


Fig. 35.10. The flowchart for a 'buzz' program.

method of counting the number of clicks is used, the click routine could also be put into the form of a subroutine which is called when wanted.

We'll illustrate the use of the DJNZ method, with the new commands added as shown in Fig. 35.11. We can do this with a little bit of editing of the code that we

```

LD B, 255
Back : PUSH BC
... click routine ...
POP BC
DJNZ, Back
RET

```

Fig. 35.11. How DJNZ can be used for two different counts. The contents of the BC registers are saved on the stack.

used for the click program, and we'll change the accumulator values to 7 and 23 at the same time so that the border colour stays white.

```

1Ø CLEAR 325ØØ: LET j = 325ØØ
2Ø FOR n = Ø TO 22: READ b
3Ø POKE j + n, b: NEXT n
5Ø DATA 6, 255, 197, 6, 255, 62, 7, 211, 254,
   16, 252, 6, 255, 62, 23, 211, 254, 16,
   252, 193, 16, 236, 2Ø1
1ØØ PRINT USR j

```

Fig. 35.12. The buzz program in POKE form.

The result is shown in Fig. 35.12. It produces a buzz which illustrates well that we got the methods correct, and we can now try altering some items. For example, if we change the 255 byte in the click routine to 128, run it, and then try 5Ø and run that we can see (or hear) how the note varies with the time delay – shorter delays give higher pitch notes.

```

1Ø CLEAR 325ØØ: LET j = 325ØØ
2Ø FOR n = Ø TO 14: READ b
3Ø POKE j + n, b: NEXT n
5Ø DATA 6, 255, 197, 6, 255, 62, Ø, 211,
   254, 16, 252, 193, 16, 244, 2Ø1
1ØØ PRINT USR j

```

Fig. 35.13. Checking if there is no buzz if only one value of bit is sent to the loudspeaker.

| | |
|---------------------|---------|
| LD B, 255 | 6,255 |
| loop: PUSH BC | 197 |
| LD A, B | 12Ø |
| exit: OUT (port), A | 211,25Ø |
| DJNZ, exit | 16,252 |
| POP BC | 193 |
| DJNZ, loop | 16,247 |
| RET | 2Ø1 |

```

1Ø CLEAR 325ØØ: LET j = 325ØØ
2Ø FOR n = Ø TO 11: READ b
3Ø POKE j + n, b: NEXT n
5Ø DATA 6, 255, 197, 12Ø, 211, 25Ø, 16, 252,
   193, 16, 247, 2Ø1
1ØØ PRINT USR j

```

Fig. 35.14. Putting numbers out from the port. Note the colour and sound effects when this runs, but disconnect your printer!

We can confirm that we need both the 1 and the Ø bits sent to the speaker by altering the routine so that it sends only one bit (try Ø or 16). You will see the border affected, but no sound. Fig. 35.13 shows the routine – remember that the displacement number for the overall DJNZ step has to be changed each time you change this number of bytes in the routine. The small loops in the click routine do not need changing.

You can create some interesting visual effects if you alter the border colour bytes each time you run the click part of the routine. The simplest way of doing this is just to load the contents of the B register that is used for the overall count into the accumulator to send to the port. Since this number changes each time, the result will be different border colours. Fig. 35.14 shows some suggestions.

Since the same port which controls border colour and the loudspeaker also controls the MIC socket, we can use it to send signals out to the cassette recorder, and it would be possible to write a routine that would send out signals in standard RS232 form (see Chapters 25 and 26) for a printer, a modem, or some other serial device. Writing an RS232 routine for the cassette port is not exactly the kind of job that is recommended for the beginner, but you can see at this stage how the essential parts of it must work. The start bit is sent out by loading a 15 into the accumulator and sending it out to port 254. The reason for using 15 is that it sets bit D3 to 1, and keeps bits D2 to DØ at 1 so that the border does not change colour. A timing loop must be used to control for how long this signal is sent out, then each bit from the selected byte has to be loaded into the accumulator and sent out. How do we do this? One method is to clear the carry flag (an OR A will do this), and then rotate the byte in the accumulator left. If a 1 is rotated out, the carry flag will be set; if a Ø is rotated out, the carry flag will be reset, so that this flag can be tested and used to call routines which put out the numbers 15 or 7 according to the bit. The delay routine will be used between each of these port output operations, and when all eight bits have been sent out (you will have to count them!), then two stop signals (1) will be sent, again using the same time delay.

This is a comparatively simple scheme, and for driving a serial printer, the parity operation can often be ignored, and many circuits can work with Ø and + 5 V signals in place of the -12 V and + 12 V signals that RS232 strictly should use.

That's the easy bit, though. The difficult bit is that the Spectrum stores its listings in compressed form, using the tokens in place of keywords, so we can't send out the bytes simply by reading through the memory. Once again, it's possible if you can find the routines in the ROM which are used for listing a program, and there is also the possibility of intercepting the signals at the printer port – but this is not beginners' work!

Chapter Thirty-six

Monitors, Assemblers, and Disassemblers

Debugging delights

Now that you have been exposed to the delights of machine code programming, the time seems ripe to talk about debugging. A bug is a picturesque name for a fault in a program, and debugging is the process of removing such flaws. There's probably a name for the programmer who put the flaws in place, but our task at present is to find how we can best deal with bugs.

The first part is prevention. Check your flowchart carefully to make sure that it really describes what you want to do, and then check your assembly language program equally carefully to make sure that it also does what you expect of it. Then check that the bytes that you are placing into memory correspond to the assembly language instructions and data. If you do all this you will eliminate quite a number of bugs before they start to crawl out of the woodwork. Don't feel that you are a failure if you don't get them all out – unless a program is very simple, there's a very good chance that there will be a bug in it somewhere, and it happens to all of us.

If you use an assembler, one very potent source of bugs disappears almost instantly. Human frailty means that the process of converting assembly language instructions into bytes in memory by looking up tables is an error-prone business, and by letting the computer tackle this, a lot of bugs can be prevented. I shall describe briefly the use of the ULTRAVIOLET © assembler program later in this chapter. If machine code really catches your imagination and you want to branch out into much more advanced work than this book can cater for, then an assembler should be a priority item.

If, however, you want to use machine code only as a way of carrying out minor tasks that are impossible or too slow for BASIC, then the POKE-to-memory method that we have used throughout this book may be perfectly adequate. It does mean, however, that there will be bugs lurking in every corner of the code. The main cause of these bugs is tedium. Converting an assembly language program into denary bytes is a tedious job, and all tedious jobs generate mistakes (ever seen a 'Friday' car?). Faulty conversion of hex to denary is one possibility, just writing down the wrong figure is another which is surprisingly common. One very potent source of trouble is in JR or DJNZ displacement values. You may get the number wrong, or you may start with them correct and then forget that if you add or delete code between the jump start and its finish, you will also need to alter the displacement. This again is a problem which is solved when an assembler is used. An incorrect jump will almost always cause the computer to lock-up or go

into its NEW routine, and unless you recorded your source routine (the BASIC program which POKEd the memory or which holds the assembler instructions), or the machine code itself, then you have lost what might have been a fair amount of effort. Another form of incorrect jump, of course, which is more difficult to spot, is doing the opposite of what you intended, such as JR Z in place of JR NZ. Careful thought about what the jump will do for various bytes should eliminate this one.

All of these problems can be overcome by meticulous checking, and it pays to be extra careful about JR displacements, and about the initial contents of registers. A very common fault is to make use of registers as if you could assume that they contained zero at the start of the program. You can never assume this – it's much safer to assume that each register will contain some value that will drive the program bananas if it is used! What do you do, however, if you have checked everything in sight, and the program completely refuses to do what you expect?

There's no simple answer to that one. It may be that your flowchart doesn't do what you expected it to, and if you didn't draw one, then you've got what you deserved. It may be that you are making use of a Spectrum ROM routine, and it doesn't operate in the way that you expect – until we have a complete breakdown of the ROM, we simply have to use trial and error. All I can do here is to give you some hints about removing the bugs from problem programs that seem to be reasonably well constructed but which don't work according to plan.

The first golden rule is never to try anything new in the middle of a large program. Ideally, your machine code program will be made up from subroutines, each of which you have thoroughly tested before assembling into a program. In real life, this is not so easy, because Spectrum cannot MERGE machine code routines, but it is still possible to load recorded code into sections of memory that are adjacent to each other, or to MERGE data lines of BASIC programs, so that more bytes can be POKEd, which is a much safer method. If you keep well tried subroutines on tape, preferably as BASIC POKE programs, then you can save a lot of programming effort by combining them, remembering that if any of the subroutines are non-relocatable, you will have to alter address bytes within the program. Once again, users of an assembler have the best of all worlds, because if the assembly language instructions are stored within REM statements, they can be MERGED and edited to your heart's content before being assembled.

Even if you do not use a subroutine library on tape, it helps to keep a note of routines. *Personal Computer World* runs a series called SUBSET, which prints several general purpose machine code subroutines each month, and most of these are Z-80 routines, reflecting the importance of this microprocessor type. Even if you don't use the routines, the way in which they are documented should give you some idea about how you are going to keep a record of your own routines and I personally think that this feature is worth my annual subscription by itself. If you are going to use a new routine in a program it makes great sense to try it out on its own first, so that you can be sure of (a) what you need to have in the registers before the program is called, and (b) what you will have in the registers after it has run.

This type of planning should eliminate bugs in a big way, but if you are still faced with a program which you don't want to start pulling apart, routine by routine, the best method of dealing with it, assuming that no good monitor program is available, is to insert breaks. A break as far as a Spectrum machine code program is concerned, is the RET instruction, coded 201. When this is encountered, the contents of the BC register are handed to the Spectrum operating system, and normal service is resumed. Now if you want to find where the problem lies in a program, the way of using a break is to pick a spot where you want to check a value. This value might be in one of the other registers, so you will have to add an instruction before the RET which will place the contents of that register or registers into the BC pair. The break bytes can then be edited into the POKE part of the program in the DATA lines, and the program assembled and tested. When the program reaches the breakpoint, it will dump a value into the BC registers and then return to BASIC leaving you to chew over the meaning of the number that appears. If the program appears to be working well as far as this breakpoint, then remove it and put another breakpoint in at a later stage. By repeating this process, you should eventually be able to find where the fault lies, and this is usually all the clue you need to find what the fault is.

The most awkward faults are faulty loops, because they almost invariably cause a lock-up, and on the Spectrum there is no way out of this. Some machines have a 'hardware reset', a button which can be pushed to restore the machine to normal operation even if it has become locked into a machine code loop. This is not available on the standard Spectrum but it does appear on other Z-80 based machines. It is likely, therefore, to be offered by one of the many independent hardware suppliers, and it would make life much easier for assembly language programmers.

One fault which can so often cause such an endless loop lock-up is a loop back to the wrong position. For example, if we had a program such as:

```
LD B, 255
Back: OUT (Port), A
DJNZ Back
```

assembled 'by hand', we may have made the DJNZ instruction loop back to the LD B, 255 instruction rather than to the OUT instruction. This will result in the B register being topped up to 255 each time the loop is executed, so that the DJNZ can never decrement B to zero, and so the loop is endless. A mistake like this is easily spotted in assembly language, because the position of the label name is easily checked, but it can be very difficult to find when you can only see the machine code bytes. Once again, taking care over loops is the only answer, and the method that has been shown in this book of writing the machine code bytes against the assembly language instructions is a very good way out of the problem.

Monitors

I mentioned monitors briefly a few pages back. It is unfortunate that the word

'monitor' has come to be used for two different items that are both associated with computers. The TV engineering definition of a monitor is of a TV-type display which accepts TV signals directly, rather than by conversion to a transmitter-type signal as is used by Spectrum (see Chapter 28). The other meaning of monitor is a program that checks (monitors) every action of the machine, and this is the type of monitor I mean in this section.

Monitors have developed over the years. In the early days of home computers, the most that could be expected of a monitor was that it would display a section of memory in hex, change memory contents to other values one byte at a time, shift a set of bytes from one starting address to another, and possibly insert breakpoints.

A few computers in days gone by had machine code monitors built in, sometimes confusingly referred to as 'front-panels', a name used in the dinosaur days of computers. Nowadays, monitors are available as programs on tape or disc which can greatly extend the usefulness of the machine for checking machine code. Some recent monitors, for example, allow a machine code program, whether in RAM or in ROM, to be run one step at a time, and will display register and memory contents (of memory locations affected by the step) at each step. A monitor of this type, of which the best known example is the Mumford STEP-80 for the TRS-80, is to a machine code programmer what a power drill is to the handyman – you start to wonder if life could continue without it.

Using an assembler

Some description of how an assembler is used is necessary in any book dealing with assembly language and machine code – it would be like describing motor maintenance with no mention of spanners if I missed this out.

An assembler is a program, usually in machine code, which has to be loaded like any other program. This is a fairly fast operation, even when cassettes are used. Once the assembler is loaded, it may present you with a menu of options. Typical of these would be entry or editing of assembly language, saving or loading of a program written in assembly language (known as source code) or one written in machine code (known as object code), assembly of source code into machine code, and movement of an editing 'pointer' which allows lines to be selectively edited. A session of code writing would start by the selection of the writing option.

Each assembler has its own peculiarities, often reflecting peculiarities of the machine on which it is used, but most Z-80 assemblers follow the standards of assembly language that were laid down by Zilog, the designers of the Z-80. Rather than deal with assemblers in general, however, it's probably more useful at this point to illustrate this section with reference to the first assembler that became available for the Spectrum, the ACS ULTRAVIOLET. For a list of the rest, see Appendix A.

The ULTRAVIOLET assembler

The ULTRAVIOLET assembler accepts address and data in denary, with no hex

needed at the writing stage, so that it is easy to use with the denary numbers that are shown in the Spectrum manual. All of the Z-80 instructions are correctly assembled, and code can be placed into memory directly (with some precautions) and if necessary relocated. Full listings of the assembly language and the code, with the curious mixture of denary for addresses and hex for codes can be displayed or sent to the ZX printer, which is particularly useful if long programs are being developed.

The style of the ULTRAVIOLET assembler, however, is very similar to that of assemblers for the ZX81, and it does not take full advantage of the ability of Spectrum to make memory space for machine code in high memory addresses. The program also requires the assembly language statements to be written inside BASIC REM lines, as was used for the ZX81, and the code is also stored in a REM line, the first line of the program. Assembled code can, however, be written in a form that will allow it to be recorded and correctly relocated to high memory when it is reloaded, with all addresses corrected. Despite the disadvantages of being modified from a ZX81 program (which was inevitable because to write an assembler from scratch is an awesome task), it is a useful piece of software.

ULTRAVIOLET, like its matching disassembler, INFRARED (I wonder if they'll make a monitor called X-RAY?), is available in 16K or 48K versions, of which I have used only the 16K version. This is on a cassette which, once I had adjusted the head angle of my cassette recorder slightly, loaded easily. The system uses a loading section, titled 'uv loader', a main piece of machine code titled 'uv 16K', and a display section ('uv 16K') which shows a few instructions – detailed instructions come on a printed sheet. When the display is complete, pressing any key will produce the switch-on display, with the copyright notice showing. This, though alarming, is perfectly normal for this program – it means that the program is in place in protected memory and ready for use.

Writing the program

The assembly language program has to be written using lower case for the assembly language instructions, and in a BASIC REM line. The first REM line should contain enough spaces, or any other characters, to leave room for the machine code which will be placed in it. Since BASIC starts at 23755, and the first four bytes of the first line are line number and length number, with the fifth byte the REM token, the machine code bytes will start at 23760. The second line of the program must contain the word 'go' following REM – this is the 'start assembly' command. The third line must contain the origin address, written as 'org', which will normally be 23760. Any attempt to assemble code into high memory, such as 32500, for example, is likely to overwrite the code of the assembler itself and cause a crash. This can be avoided by using a modification of the 'org' statement, which places the relocation address in brackets after the 'org' address. For example, using org 23760(32500) will cause code to be assembled at 32760, but in a form which will run correctly when the bytes of code are shifted to 32500 (this can mean that it would not run at 23760!). The relocation can then be performed by saving the code

and then loading it into the higher address. You are not restricted to a single 'org' statement, and the assembler can be used to produce a number of machine code sections which are located at different addresses.

Following the 'org' statement, the assembly language program can be typed in, following the rules of the ACS system. Each line must start with a REM, following which the assembly language can be typed using *lower case letters*, as used in the Spectrum manual, for the instructions. Label names *must start with a capital* (upper case) letter, and can be of any length. It's best, however, to stick to short names to save memory, particularly on a 16K machine which does not have much memory left when the ULTRAVIOLET assembler has been loaded in – the length of the code is about 5000 bytes. The *close-bracket* and + characters must not be used in a label name, otherwise there are no restrictions. The separating mark between a label and the assembly language following it is a *semicolon*.

Comments have to be preceded by an exclamation mark rather than by the semicolon which is normally used in Z-80 assembly language work. Up to 32 characters of comment can be printed on a line; if more are needed, then a new REM line must be used. Several assembly language statements can be written in a single assembly language line following a single REM – it appears that unless your program is very large, you could place all of it in one line! The statements must, however, be separated by semicolons. This is essential because if a colon is used as it would be in BASIC, the Spectrum operating system will then treat the next key as signifying a keyword rather than a letter.

The end of the assembly language is signalled by another REM line which contains the word 'finish'. The BASIC program can then be checked and edited in the usual way, and when you are satisfied that it looks correct, assembly can start. This, for the 16K version, means typing RANDOMIZE USR 27500 – the figure for a 48K version is 60000 – which if all is well will produce the program code on the screen. The display shows the assembly language and the generated machine code, with addresses in denary and code in hex, so that if you want to copy the bytes to use in a BASIC POKE program, you will have to convert them for yourself. The assembler runs twice, causing the screen to seem to flicker when the program is a short one. The reason is that on the first run through, the assembler may find label names to which it can't allocate an address, because the addresses for these labels come later. They are noted, and on the second run through all such 'forward references' are correctly allocated. A short program will occupy less than one screen, and if a printout is needed, the COPY key of Spectrum has to be used. When the assembled code requires more than one screen, assembly will stop at the bottom of a screen, and will not continue until a key is pressed. On the second run, when a screen is filled, pressing the 'p' key will cause that screen to be printed. This is necessary because Spectrum facilities like the COPY key cannot be used while the assembler program is operating.

Error messages

Errors are very well detected and pointed out. If there is a fault in 'go', 'org', or

'finish' statements, an error message will be printed instead of starting assembly. If the fault is in the assembly language syntax, such as writing 1 da, 12 in place of 1 d a, 12, then assembly will stop at the faulty line, and the error message will show in detail which statement in the BASIC REM program is faulty. There will also be a Spectrum operating system error message, which will be "B integer out of range", or "2 variable not found" or 'Q parameter error', but these messages are less important.

Useful features

As well as permitting the 'org' instruction (strictly speaking, this is a *pseudo-instruction*, meaning that it does not assemble into code), the ULTRAVIOLET permits four others of this type. One is equ, which can specify an address for a label name in advance. For example, the line:

30 REM equ 23560 Loop

will ensure that the address 23560 is filled in wherever the word Loop is used as an operand. This is very useful if a program has to be written in several versions (such as for 16K or 48K, for example), as all the addresses can be altered just by changing a few equ statements. Another use suggested by the program writers of ULTRAVIOLET is in linking separate sections of code so that they will run as one, as the 16K version of ULTRAVIOLET does.

Other useful pseudo-instructions are defb, defw and defs, which allow a form of POKE operation. Using defb will place the single-byte number that followed defb into memory at the address to which the statement is assembled; defw allows two bytes to be placed into memory, low byte first in this fashion. The defs pseudo-instruction is one of the most useful of this group, because it allows a string of ASCII codes to be stored by specifying the letters. For example, defs Sinclair will store the eight ASCII codes for the name Sinclair.

Other features and summary

It is possible to write code that will change addresses or data by itself – self-modifying code. By labelling a load instruction with a label word, for example, the bytes following the load instruction (byte to be loaded or address bytes) can be changed by a later piece of code which loads to the address label + 1 or label + 2. This is rather more advanced programming than we can cover in this book, but it is a very useful feature of the assembler.

Once its features, which seem odd to anyone who has used other assemblers, have been mastered, ULTRAVIOLET is a useful program. It is not, however, in the same class of assembler as the superb ZEN, which is used on TRS-80, Nascom, and Sharp machines. ULTRAVIOLET will, however, be a program with which ZX81 machine code programmers will feel at home, and it is a useful tool, particularly for the 48K Spectrum, which has much more memory to play with. The demonstration program in Figs 36.1 and 36.2 gives some idea of how

```

10 REM
20 REM go
30 REM org 23760
40 REM xor a
50 REM ld de, Addr
60 REM call 3082
70 REM set 5, (iy + 2)
80 REM call 5588
90 REM ret
100 REM Addr; defb 128
105 REM defs, Push any key ...
110 REM defb 174
120 REM finish

```

Fig. 36.1. How assembly language instructions are written in the ULTRAVIOLET assembler. This program can be saved by normal BASIC methods. It is assembled by calling the assembler – for the 16K version this means typing RAND USR 27500.

```

org 23760
23760 AF xor a
23761 11 DF 5C ld de, Addr
23764 CD 0A 0C call 3082
23767 FD CB 02 EE set 5, (iy + 2)
23771 DC D4 15 call 5588
23774 C9 ret
Addr
defb 128
defs Push any key
defb 174

```

Fig. 36.2. The appearance of the assembled code on the screen. The program prints a message and waits for a key to be pressed.

ULTRAVIOLET can be used, and the type of printout that it produces. Another assembler, from Abersoft, uses the Microdrives.

Now that you have some idea how to use monitors, assemblers and disassemblers, and you know some of the basic theory of machine code, you're ready to go on to more advanced books on the subject – books that assume you're familiar with everything we've been talking about! This section hasn't covered all you need to know about machine code, but I hope it has given you enough basic information to carry on for yourself, and to use the available software for your own research and experiments. Above all, I hope it has made the machine code section of the Spectrum manual a little easier to understand! In Appendix D you'll find a full list of Z-80 assembly language instructions and codes in alphabetical order – rather easier to use than the list in the manual. Although we have covered only a few of them here, further work with more advanced books should

familiarise you with the others you need to know, and I hope that this short introduction has taken some of the mystery out of machine code.

Part 7
Appendices

Appendix A

Selected List of Spectrum Software

Compiled by Phil Gardner

A complete list of Spectrum software would fill a book by itself, so to keep this one down to a size that the average Spectrum owner can actually lift we've only listed a representative selection of some of the best and most readily available software. The table that follows lists some 200 programs in alphabetical order under broad categories like 'Adventure Games' and 'Utilities'. Inevitably we've left out some good programs, mainly because good programs are being released every week and we had to go to press some time! Prices and other information were correct when this book went to press, but may have changed since, so do check out the magazine advertisements or ask your local dealer before ordering.

| Program | Publisher | Price | Comments |
|------------------------|--------------------|--------|---|
| Adventure Games | | | |
| Colossal Adventure | Level 9 | £9.90 | Complex textual adventure. |
| Espionage Island | Artic Computing | £6.95 | Find the secret of the island. |
| Greedy Gulch | Phipps Associates | £4.95 | Set in Western mining town. Good graphics and descriptions. |
| Inca Curse | Artic Computing | £6.95 | Recover treasure from Inca temple: enlivened by humour. |
| Oh Mummy! | Gemini | £4.95 | Treasure quest with a surprise twist. |
| Pimania | Automata | £10.00 | Adventure quest with real prize worth £6000. |
| Planet of Death | Artic Computing | £6.95 | Find your captured spaceship. |
| Quest Adventure | Hewson Consultants | £5.95 | Complex adventure with graphics and puzzles. |
| Quetzalcoatl | Virgin Games | £5.95 | 3D graphic adventure with unpronounceable but exciting dangers. |
| The Castle | Bug Byte | £6.95 | Entertaining fantasy adventure. |
| The Golden Apple | Artic Computing | £6.95 | Island treasure hunt with colour graphics and sound. |
| The Hobbit | Melbourne House | £14.95 | Superb addictive adventure with graphics; based on Tolkien. |
| Urban Upstart | Richard Shepherd | £6.50 | Adventure in seedy urban setting. Witty. |

| Program | Publisher | Price | Comments |
|---------------------|------------------|--------|---|
| Valhalla | Legend | £14.95 | Superb moving graphics complement this addictive Norse adventure. |
| Woods of Winter | CRL | £5.95 | Escape winter: seek sanctuary in the castle. |
| Arcade games | | | |
| 3D Ant Attack | Quicksilva | £6.95 | Excellent 3D maze rescue game. |
| 3D Combat Zone | Quicksilva | £5.95 | 3D Martian battlefield with tanks and aliens. |
| 3D Desert Patrol | CRL | £4.95 | Lead your patrol out of desert minefield. |
| 3D Spawn of Evil | dk'tronics | £4.95 | Destroy the breeding vermin – an unusual and entertaining game. |
| 3D Tanx | dk'tronics | £4.95 | Destroy Rommel. Good 3D graphics. |
| 3D Tunnel | New Generation | £5.95 | Superb animated graphics for the many hazards. |
| Ah Diddums | Imagine Software | £5.50 | Teddy bear escapes from toy box. Superb animated graphics. |
| Alchemist | Imagine Software | £5.50 | Good adventure/arcade game. |
| Android One | Vortex Software | £5.95 | Guide laser-bearing android to reactor and defuse it. |
| 'The Reactor Run' | Vortex Software | £5.95 | Varied, with good 3D effects. |
| Android Two | Vortex Software | £5.95 | Eat and enjoy, but beware the rats. |
| Apple Jam | dk'tronics | £4.95 | You guide a speedboat and waterskier through many obstacles. |
| Aquaplane | Quicksilva | £6.95 | Seek and destroy underwater death machines. |
| Aquarius | Bug Byte | £5.95 | Waves of different aliens attack. Well presented. |
| Arcadia | Imagine Software | £5.50 | Good fast Galaxians clone. |
| Astro Blaster | Quicksilva | £4.95 | Good arcade/adventure blend with 3D graphics. |
| Atic Atac | Ultimate PTG | £5.50 | Blast the aliens in this maze pursuit game. |
| Bedlam | M. C. Lothlorien | £5.95 | Good, though not what you might expect! |
| Birds and Bees | Bug Byte | £5.95 | Good version of Centipede. |
| Centi-bug | dk'tronics | £4.95 | Original and bizarre cake-baking game. |
| Cookie | Ultimate PTG | £5.50 | Search and destroy computer in maze of corridors. |

| Program | Publisher | Price | Comments |
|--------------------------------|-------------------|-------|--|
| Cosmic Debris | Artic Computing | £4.95 | Excellent version of 'Asteroids'. |
| Dimension Destroyers | Artic Computing | £5.95 | Good 3D space battle game. |
| Doomsday Castle | Fantasy Software | £6.50 | Find your way through the castle and collect stones and crystals. |
| Electro Storm | PSS | £5.95 | Very close to arcade quality version of 'Defender'. |
| Escape | New Generation | £4.95 | Find axe and escape from random maze, pursued by dinosaurs. |
| Exterminator | Silversoft | £5.95 | Faithful version of Robotron. |
| Frogger | A & F Software | £5.75 | Excellent version of arcade classic. |
| Galaxians | Artic Computing | £5.95 | Excellent copy of arcade original. |
| Gulpman | Campbell Systems | £5.95 | Good Pacman clone, with 15 mazes. |
| Haunted Hedges | Micromega | £6.95 | 3D Pacman maze chase. |
| Hopper | PSS | £5.95 | Good version of Frogger. |
| Horace and the Spiders | Psion | £5.95 | Clear Spider Mountain of spiders. Not for the phobic! |
| Hungry Horace | Psion | £5.95 | Entertaining maze chase set in a park. For younger players. |
| Invaders | Artic Computing | £4.95 | Good version of this arcade classic. |
| Invasion of the Body Snatchers | Crystal Computing | £6.50 | Superior version of Defender. |
| Jet Pac | Ultimate PTG | £5.50 | Superb space trading and battle game. |
| Jogger | Severn Software | £4.95 | Human frogger. |
| Joust | Softek Software | £5.95 | Unusual jousting contest with good animation. |
| Jumping Jack | Imagine Software | £5.50 | Jumping through gaps in levels on the screen. More fun than it sounds. |
| Jungle Fever | A & F Software | £6.95 | Good combination of fast-moving graphics and adventure. |
| Knot in 3D | New Generation | £5.95 | Mind-boggling 3D flight, avoiding your trail. Unusual; excellent. |
| Kong | Ocean Software | £5.90 | Climb through hazards to rescue beautiful (?) girl from Kong. |
| Kosmic Pirate | Elephant Software | £5.65 | For once you're the baddie. |
| Light Cycle | PSS | £4.95 | Fast, responsive version of this game from Tron. |
| Lunar Jetman | Ultimate PTG | £5.50 | Excellent, varied game – deliberately no instructions! |

| Program | Publisher | Price | Comments |
|------------------|--------------------|-------|---|
| Manic Miner | Bug Byte | £5.95 | Miner Willy escapes from 20 locked caverns. |
| Maziacs | dk'tronics | £4.95 | Large, complex maze game. |
| Megapede | Softex Software | £5.95 | Good version of 'Centipede'. |
| Missile Defence | Anirog | £5.95 | Familiar type, with 'smart bombs'. |
| Molar Maul | Imagine Software | £5.50 | Brush your teeth to stop tooth DK (decay)! |
| Mr Wimpy | Ocean Software | £5.95 | A Burgermania clone. |
| Mugsy | Melbourne House | £6.95 | Comic-strip animated graphics: blend of arcade, adventure and simulation. |
| Painter | A & F Software | £5.75 | Aerosol paint-squirting in maze chase! |
| Panic | Mikrogen | £5.95 | Dig holes in scaffolding and trap aliens. |
| Penetrator | Melbourne House | £6.95 | Exceedingly good version of Scramble. |
| Pheenix | Megadodo Software | £5.50 | Destroy waves of giant robotic birds. |
| Pssst | Ultimate PTG | £5.50 | Robbie uses insect sprays to protect flower from ravenous pests. Great! |
| Pyramid | Fantasy Software | £5.50 | Explore the chambers and zap the aliens. |
| Road Toad | dk'tronics | £4.95 | Frogger type. |
| Rommel's Revenge | Crystal Computing | £6.50 | Good if slow version of Battle Zone. |
| Sam Spade | Silversoft | £5.95 | Good version of Space Panic (dig holes to trap aliens). |
| Slap Dab | Anirog | £5.95 | Painting while pursued by woodworm. |
| Spectres | Bug Byte | £8.00 | Good variant of Pacman. |
| Splat | Incentive Software | £5.50 | Addictive, unusual maze game – the maze moves and the perils get more perilous. |
| Star Trek | Gemini Software | £5.95 | Full version of the classic game. |
| Starfire | Virgin Games | £7.95 | Space battle. |
| Timegate | Quicksilva | £6.95 | Classic space battle game, excellent despite illiterate text. |
| Tranz Am | Ultimate PTG | £5.50 | Excellent racing/fighting game. |
| Wild West Hero | Timescape | £5.50 | Excellent, fast version of Robotron. |
| Winged Avenger | Work Force | £5.00 | Good version of the arcade game 'Phoenix'. |

| Program | Publisher | Price | Comments |
|--------------------------|------------------|--------|---|
| Xadom | Quicksilva | £6.95 | Retrieve an Artefact. Blend of arcade and adventure. |
| Zzoom | Imagine Software | £5.50 | Unusual shoot-it-down game. |
| Board games | | | |
| Automonopoli | Automata | £6.00 | Good version of Monopoly. |
| Backgammon | Psion | £5.95 | Plays a strong game, and has good features. |
| Chess | Artic Computing | £8.45 | Seven levels, colour and sound. |
| Chess: the Turk | OCP | £8.95 | Excellent chess program. |
| Draughts | CRL | £4.95 | Fast, competent play. |
| Scrabble | Psion | £15.95 | Excellent. Plays a strong game, though the computer challenges some simple words. |
| Business programs | | | |
| Cash Book | Gemini | £59.95 | For cash-based small business; needs printer. |
| Accounting | Gemini | £19.95 | Sales, purchase and nominal ledgers. |
| Easiledger | Gemini | £59.95 | Needs Gemini 'Cash Book Accounting' – takes data and gives final accounts. |
| Final Accounts | Gemini | £8.95 | Home and business accounting. |
| Graphplot | Gemini | £19.95 | Takes data from other Gemini programs and plots graphs. |
| Invoicing | Transform | £15.00 | Needs printer; handles discounts and VAT, etc. |
| Masterfile | Campbell Systems | £15.00 | Superb database, flexible, full of good features. |
| Payroll | Transform | £19.95 | Handles up to 40 employees. |
| Plus 80 Address Manager | OCP | £19.95 | Fast, professional address database; 80-column output. |
| Purchase Day Book | Transform | £12.75 | 11–15 subheadings; handles VAT. |
| Purchase Ledger | Kemp | £14.95 | All standard accounting and VAT facilities. |
| Sales Day Book | Transform | £12.75 | Invoices, statements, VAT, etc. |
| Sales Ledger | Kemp | £14.95 | Corresponds to Kemp's 'Purchase Ledger': similar facilities. |
| Stock Control | Kemp | £14.95 | Standard facilities. |
| Stock Control | Gemini | £19.95 | Standard facilities. |
| Stock Control | Transform | £12.75 | Standard facilities. |
| Tasword Two | Tasman Software | £13.90 | Excellent professional-standard word processor. |

| Program | Publisher | Price | Comments |
|---------------------|--------------------|-------|--|
| Vu-Calc | Psion | £8.95 | Spreadsheet program. |
| Education | | | |
| A,B,C,... Lift Off! | Longman Software | £7.95 | Reading and language development game. |
| Ballooning | Heinemann Software | £9.95 | Teaches principles of hot-air ballooning, plus balloon flight simulator. |
| Ephemeris | Bridge Software | £7.90 | Calculates planetary positions. |
| Forensic | AVC | £3.00 | Sherlock Holmes game for 'O' level chemistry students. |
| Fractions 1 | Kemsoft | £5.95 | Teaching fractions with good graphics, and tests. |
| Human 1 | AVC | £3.00 | Hangman with biology topics at CSE/'O' level. |
| Jungle Maths | Scisoft | £6.95 | Arithmetic games. |
| Lunar Letters | Longman Software | £7.95 | Game to teach letter recognition. |
| Masterchem | Cloud 9 | £4.50 | 'Mastermind' quiz on chemical elements and their symbols. |
| Pascal | Chalksoft | £6.95 | Well presented exposition of Pascal's Triangle, with questions. |
| Pathfinder | Widgit Software | £5.95 | Maze games for exploring counting and directions. |
| Quick Thinking | Mirrorsoft | £6.95 | Several games for practising arithmetic. |
| Robot Runner | Longman Software | £7.95 | Good arcade-type game involving solving multiplication problems. |
| Statistics | Bridge Software | £9.90 | Statistical calculations, tests, etc. |
| Tense French | Sulis Software | £9.95 | Revision/learning tool on French verbs and tenses. |
| Young Learners 1 | Rose Software | £4.95 | 4 programs for primary children: counting, time, money, etc. |

Games (miscellaneous)

| | | | |
|---------------|-----------------|-------|--|
| 3D Quadracube | Artic Computing | £4.95 | 3D, four-in-a-row noughts and crosses. |
| Bridge Player | CP Software | £8.95 | Sophisticated bridge game, with different bidding conventions. |
| Dictator | dk'tronics | £4.95 | How long can you stay president of Ritimba? |
| Forensic | AVC | £3.00 | Sherlock Holmes game for 'O' level chemistry students. |

| Program | Publisher | Price | Comments |
|-------------------------------|----------------------------|--------|--|
| Fruit Machine | dk'tronics | £4.95 | Nudge, hold, etc. – but how do you win the money? |
| Gangsters! | Cases Computer Simulations | £6.00 | Can you take over the underworld? |
| Human 1 | AVC | £3.00 | Hangman with biology topics at CSE/'O' level. |
| Jumbly | dk'tronics | £6.95 | Unjumble the pictures by moving sliding blocks around. |
| Ringo | Elephant Software | £5.00 | Intriguing variant on the Rubik Cube idea. |
| Languages (Computer) | | | |
| Forth | Abersoft | £14.95 | Good version of extended FIG-Forth. |
| Logo | Kuma | £9.95 | The 'Turtle' graphics language. |
| Logo | LCSI/SOLI | £39.95 | Full implementation |
| Micro-PROLOG | LPA | £24.95 | Artificial intelligence language: excellent for databases. |
| Pascal 4T | Hisoft | £25.00 | Virtually full implementation of standard Pascal. |
| Miscellaneous programs | | | |
| Cattel IQ Test | Sinclair Research | £12.95 | Standard IQ test, proving you're a genius?? |
| The Computer Cook Book | Bug Byte | £9.50 | Recipes, menu selection, ingredients, wine choice, etc. |
| Practical programs | | | |
| Comp-U-tax | Micromega | £9.95 | Calculate your personal income tax. |
| Finance Manager | OCP | £8.95 | Home (and business?) accounting. |
| Home Accounts | Gemini | £19.95 | Comprehensive home accounting package. |
| Masterfile | Campbell Systems | £15.00 | Superb database: flexible, full of good features. |
| Personal Banking System | Micromega | £9.95 | Handles payments, receipts, balance, standing orders, etc. |
| Plus 80 Address Manager | OCP | £19.95 | Fast, professional address database; 80-column output. |
| Statistics | Bridge Software | £9.90 | Statistical calculations, tests, etc. |
| Tasword Two | Tasman Software | £13.90 | Excellent professional-standard word processor. |
| Vu-Calc | Psion | £8.95 | Spreadsheet program. |

| Program | Publisher | Price | Comments |
|-------------------------|----------------------------|--------|---|
| Simulation games | | | |
| 1984 | Incentive Software | £5.50 | Economic/political simulation. Can you run the country? |
| Apocalypse | Redshift | £9.95 | Massive, complex war-game simulation. |
| Ball by Ball | Video Software | £5.00 | Sophisticated cricket simulation (text only). |
| Battle of Britain | Microgame Simulations | £5.95 | Strategic war-game. |
| Blackjack | Chipmunk Software | £6.00 | Blackjack casino simulation game. |
| Chequered Flag | Sinclair Research | £6.95 | Good Formula One racing simulation. |
| Conflict | Martech Games | £11.95 | Strategic/economic war-game with board, etc. Sophisticated. |
| Cricket captain | Allanson Computing | £4.95 | Planning a cricket match, with some 3D animation. |
| Flight Simulation | Psion | £7.95 | Excellent and detailed flight simulation. |
| Football Manager | Addictive Games | £6.95 | Simulation: get your team from Division 4 to the FA Cup. |
| Golf | Virgin Games | £7.95 | Golf 'simulation'. |
| Golf | Abrasco | £6.95 | Golf 'simulation'. |
| Nightflight 2 | Hewson Consultants | £7.95 | Excellent flight simulator. |
| Pinball Wizard | CP Software | £5.50 | Pinball simulation. |
| Pool | Bug Byte | £5.95 | Good simulation/adaptation of pool. |
| Pool | CDS | £5.95 | Pool simulation. |
| Print Shop | Cases Computer Simulations | £6.00 | Good simulation for the budding small businessman. |
| Road Racer | Thorn EMI | £6.95 | Fast motor racing simulation. |
| Speed Duel | dk'tronics | £5.95 | Formula One racing simulation. |
| The Forest | Phipps Associates | £9.95 | Excellent simulation of orienteering, with very full documentation. |
| Trader | Pixel Productions | £9.95 | Trading through the galaxy – survive and make a profit? |
| Tyrant of Athens | M. C. Lothlorien | £5.50 | Strategic war-game in 5th-century BC Greece. |
| Utility programs | | | |
| Beta BASIC 1.8 | Betasoft | £11.00 | Toolkit with many extra keywords and functions. |

| Program | Publisher | Price | Comments |
|----------------------------------|--------------------|--------|--|
| Clone | Work Force | £5.00 | Makes back-up copies of most programs. |
| Compiler Super C | Softek Software | £9.95 | BASIC compiler; cannot handle strings, arrays or floating point. |
| Complete BASIC Programmer | Softek | £9.95 | Toolkit plus graphics kit. |
| Complete Machine Code Programmer | Softek | £12.95 | SOFSEM Editor/Assembler plus SOFMON Monitor/Disassembler. |
| Devpac 3 | Hisoft | £14.00 | Powerful, sophisticated editor/assembler plus disassembler/monitor. |
| Dungeon Builder | Dream Software | £9.95 | Adventure game designer, with graphics. Excellent. |
| Editor/Assembler | Picturesque | £8.50 | Simple assembler. |
| FP Compiler | Softek | £19.95 | Almost complete BASIC compiler, including floating point. |
| HURG | Melbourne House | £14.95 | Arcade game designer. |
| Infrared | ACS Software | £6.75 | Disassembler. Complements 'Ultraviolet'. |
| IS Compiler | Softek | £9.95 | Integer BASIC compiler. |
| Mcoder II | PSS | £9.95 | Excellent BASIC compiler, handles most integer commands and strings. |
| Melbourne Draw | Melbourne House | £8.95 | Very powerful graphics aid. |
| Micro-print 42.51 | Myrmidon Software | £5.00 | Gives 42-character or 51-character line modes. |
| Monitor | Picturesque | £7.50 | Simple monitor. |
| Programmer's Dream | Work Force | £6.95 | Advanced toolkit. |
| Scope II | The Games Designer | £17.95 | Arcade game designer with sprite routines. |
| Spectrographics | Bridge Software | £6.90 | Useful graphics generator. |
| Spectrum Machine | OCP | £9.95 | Monitor plus machine code tutor. |
| Code Test Tool | | | |
| Spectrum Master | OCP | £9.95 | Toolkit with 28 new commands. |
| Tool Kit | | | |
| Spectrum Sprites | Work Force | £5.00 | Sprite graphic generator. |
| Taswide | Tasman Software | £5.50 | Excellent utility to provide 64-character line mode. |
| The Games Designer | Quicksilva | £14.95 | Design your own arcade game. |
| The Quill | Gilsoft | £14.95 | Adventure game generator/designer. Excellent. |
| Ultraviolet | ACS Software | £7.50 | Assembler. See also 'Infrared'. |

| Program | Publisher | Price | Comments |
|------------------------|----------------|--------|---|
| User-to-User | OEL | £5.70 | Enables VTX5000 modem users to communicate and send programs. |
| Vu-3D | Psion | £9.95 | Graphics utility. Creates 3D images with shading and perspective. |
| White Lightning | Oasis Software | £14.95 | Impressive graphics package for games design. |
| White Noise & Graphics | Gilsoft | £5.95 | Adds new graphics and sound commands. |
| ZEN Editor/Assembler | Kuma Computers | £12.50 | Full Zilog Z-80 assembler. |

Appendix B

Getting the Most from Mail Order

by *Mike Scott Rohan*

And now a word about buying by mail order. *Don't!*

No, I'm not just trying to be funny. Mail order businesses provide a very useful service, and there are times when you will have no option but mail order if you have decided on a particular unit. But before you cheerily write and mail your cheque, make sure you know how to protect yourself from the don't-cares, the incompetents, and the genuine crooks.

The 'don't-cares' are basically honest traders. You'll get what you ordered – eventually. But you may have ordered on the basis of an advertisement for a product that was still on the drawing board, or at prototype stage – and because the manufacturer has got his sums wrong, or tried to stop a competitor cutting away the market for his proposed new product, you are left waiting months and months for something that isn't even properly in production. Some famous names have adopted this policy, and although the products may be worth the wait, it does create a dangerous precedent for the smaller fry. They know you'd rather get the good price – and the good product – than go to all the bother of trying to get your money back. So:

Phone and check that they are still in business and still selling the same product at the same price with supplies still in stock.

In your ordering letter, set a definite time limit for delivery (if they insist on their coupon, tack it on as well). The limit should be 28 days at the most.

Check at least once before the end of your deadline. If nothing has appeared, send a registered letter demanding your money back. Often your order will miraculously appear.

The point is that when you make an order in writing you're negotiating a contract, and if the company takes your money without argument they're accepting your terms. So if the goods don't materialise, and your money doesn't either, talk to your local Citizen's Advice Bureau, the Office of Fair Trading, the Advertising Standards Agency (if the firm is a large one) and – most importantly – the magazine that carried the advertisement, as you should be covered by their mail order protection scheme. This is because you may be dealing with one of the other two categories – an incompetent, or a crook.

Home computing has revived the cottage industry, and there are now hundreds of firms operating from garden sheds and back-street terraces. This is all very good for the economy, but many of these honest folk are not business people, and they have a charming but annoying habit of losing your order somewhere in last week's

laundry basket. At least they aren't cashing the cheque and making free with the interest, but if in doubt, badger – especially through the magazines that carry their ads. You may be driving them to a nervous breakdown, but then no one forces them to go into business in the first place. Also, there is just a chance that this kind of well-meaning incompetence is the cover for category number three – the out-and-out crook.

The only way to win with the crooks is to make sure that they never get your money in the first place. Find out all you can about a company before you send your money: has it got a proper address and phone number? If not, forget it. How long has it been trading? Have any magazines had complaints about it? The cost of a few calls could save you a lot of money.

Going bust

It happens – and when it does, you are going to be a long way down the creditor's list. So how can you protect your money?

The best way is to use a legal form of words devised by the National Federation of Consumer Groups (12 Mosley Street, Newcastle-upon-Tyne) who will sell you printed stickers saying:

This money is sent on condition that you will hold it as a trustee on my behalf, and that it will remain mine until the goods have been sent to me. If you accept this payment you will be deemed to have accepted this condition.

It won't stop thieves stealing the money, or frauds misusing it, but it may make them think twice. Their worst enemy, after all, is the person who has the sense to stand up and complain. Another and perhaps better way is to order with a credit card such as Access or Barclaycard. These companies will protect their money – and yours – with an iron fist, but you could have a lot of trouble getting the bad order wiped off your statement!

Support

Lack of easy servicing and back-up is another problem with mail order. The company may be good and offer an excellent guarantee, but it's a long way to go if and when something *does* go wrong. And if you're dealing with incompetents, they may not even have thought of support when marketing their product. However, if something you've bought doesn't perform as advertised, or at all, you're entitled to an instant refund. The law says that anything sold must be fit for the purpose it was sold for. And if it does any harm to your system as a result, you're entitled to damages.

Remedies

Your best friend in any mail order case ought to be the magazine that carried the advert. They make money from it, as the law has recognised with the protection

scheme – but note that it doesn't seem to apply to classified ads, or if the ad invites you to send for a brochure from which you make your order. If the magazine can't help you try some of the organisations mentioned earlier – and don't be afraid to use the law. The Small Claims procedure is designed for ordinary folk like us, and the more we use it, the better. The more we sit back, the more trusting we are, the harder we make it for good, honest traders to succeed.

Appendix C

A Short Add-on Atlas

This section gives the names, addresses and main products of selected add-on manufacturers and suppliers, in case you've lost the magazine where you saw that tempting new piece of hardware that you simply can't live without. Again, this information was correct at the time of going to press, but in a volatile market like this one things change very quickly, and a growing computer firm can sometimes be as hard to find as the one that's actually gone out of business!

ADS

8 Bonchurch Street, Portsmouth, Hants PO4 8RY
0705 823825
Centronics Interface

Advanced Memory Systems

Green Lane, Appleton, Warrington WA4 5NG
0925 62907
Lo-Profile keyboard

AGF

26 Van Gogh Place, Bognor Regis, W. Sussex PO22 9BY
0243 823337
Joystick interfaces

Basicare Microsystems Ltd

12 Rickett Street, London SW6 1RU
Basicare expansion system
Under new management: address likely to change very shortly.

Bi-Pak

P.O. Box 6, Ware, Herts
0920 3442
Sound synthesiser

Blackboard Electronics

17 Beechfield Road, Davenport, Stockport, Cheshire SK3 8SF
061 487 2508
A/D converter and sensors

British Micro

Unit Q2, Penfold Works, Imperial Way, Watford WD2 4YY
0923 48222
Grafpad graphics tablet

Cambridge Computing

1 Ditton Walk, Cambridge CB5 8QZ
0223 214451
Prog. joystick interface

Cambridge Microelectronics

1 Milton Road, Cambridge CB4 1UY
0223 314814
EPROM products

CCS

P.O. Box IW9, Leeds LS16 6NT
0532 670625
Joysticks and interfaces

Cheetah

24 Ray Street, London EC1
01 240 7989
Speech unit, plug-in memory expansion

Consumer Electronics Ltd

Failsworth, Manchester M35 0HS
061 682 2339
Joysticks (incl. JoySensor)

Currah Computers

Greythorpe Industrial Estate, Hartlepool, Cleveland TS25 2DF
Currah Microspeech

Custom Cables International

Units 2-4, Shire Hill Industrial Estate, Saffron Walden, Essex
CB11 3AQ
0799 25014
Lightpen, joystick interface

Datel Electronics

Unit 8, Fenton Industrial Estate, Dewsbury Road, Fenton, Stoke-on-Trent
0782 273815
Joystick, joystick interface, speech unit/interface

DCP Microdevelopments

2 Station Close, Lingwood, Norwich NR13 4AX
0603 712482
S-pack speech unit

Dean Electronics

Glendale Park, Fernbank Road, Ascot, Berks.
0344 885661
Alphacom ZX-compatible printer

dk'tronics

Unit 6, Shire Hill Industrial Estate, Saffron Walden, Essex CB11 3AQ
0799 26350
Keyboard, joystick interface, lightpen, memory upgrade

Downsway Electronics Ltd

Depot Road, Epsom, Surrey KT17 4RJ
Memory expansion, prog. joystick interface

East London Robotics

Gate 11, Royal Albert Docks, London E16
01 474 4430
Trickstick, joystick interface

Elinca Products Ltd (ZX Tapeloader)

Lyon Works, Capel Street, Sheffield 6
0742 756728
ZX tape loader

EPROM Services

3 Wedgewood Drive, Roundhay, Leeds LS8 1EF
0532 667183
EPROM board, auto-start unit

Euro Electronics Ltd

26 Clarence Square, Cheltenham, Gloucestershire
0242 582009
ZX Lprint printer interface

Fox Electronics Ltd

35 Martham Road, Hemsby, Norfolk
0493 732420
Prog. joystick interface, memory upgrade

Fuller Microsystems

71 Dale Street, Liverpool 2

051 236 6109

Keyboards, sound and speech units

Glanmire Electronics

Westley House, Trinity Avenue, Enfield EN1 1PH

Time controller

Grant Design

Bank House, Repsham, Norwich, Essex

0603 870852

Mechanical joystick

Griffin & George

Ealing Road, Wembley HA0 1HJ

01 997 3344

Input/output devices

Harley Systems Ltd

The Pepperboxes, Great Missenden, Bucks HP16 9PR

024028 630

Input/output devices and A/D converter

Hilderbay

8-10 Parkway, London NW1

01 485 1059

Printer interface, cassette recorders and accessories

ITL Kathmill Ltd

The Old Courthouse, New Road, Chatham, Kent ME4 4QJ

0634 815464

Disk interface

Jiles Electronics

48 Parkway, Chellaston, Derby DE7 1QA

0332 703892

Joystick interfaces and CO-DER tape loader

Kelwood Computer Cases

Downs Row, Moorgate, Rotherham S60 2HD

0709 63242

Sound boosters, cassette interfaces, and power switches

Kempston Microelectronics Ltd

Unit 30, Singer Way, Woburn Rd Ind. Est, Bedford MK42 7AF

0234 852997

Joystick interface and joysticks

Maplin Electronics Supplies

Maplin Complex, Oak Road South, Benfleet, Essex SS7 2BB

0702 552911

Keyboard and modem

Micro Power

8 Regent Street, Chapel Allerton, Leeds LS7 4PE

0532 683186

Add-on sound unit

Micro-Pad

14 Brackley, Queen's Road, Weybridge, Surrey KT13 0BJ

0932 42882

Joystick interface

MicroMyte Communications

Polo House, 27 Prince Street, Bristol 1

0272 299373

Acoustic coupler

Microvitec Ltd

P.O. Box 188, Futures Way, Bolling Road, Bradford BD4 7TU

0274 390011

Cub monitor

Minor Miracles

Ipswich: 0473 50304

Modem

Miracle Systems

6 Armitage Way, Kings Hedges, Cambridge CB4 2UE

Monitor interface

Morex Peripherals Ltd

172B King's Road, Reading, Berks RG1 4EJ

0734 584238

Disk and printer interfaces

Ness Micro Systems

100 Drakies Avenue, Inverness IV2 3SD
0463 790368

Cassette deck controller

Network Computer Systems

39 Bampton Road, Luton LU4 0DD
0582 508616
Multiload and multisave networking systems

Orion Data

3 Cavendish Street, Brighton BN2 1RN
0273 672994
Speech recognition unit

Orme Electronics

2 Barripper Road, Camborne, Cornwall TR14 7QN
0209 715034
EPROM boards

Petron Electronics

1 Courtlands Road, Newton Abbot, Devon TQ12 2JA
0626 62836
Sound unit

Pinnacle Ltd

Pinnacle House, Oasthouse Wharf, Orpington, Kent
0689 27000
BEEP amplifier

Primordial Peripherals

89 Herne Road, Bushey, Herts WD2 2LP
01 950 9533
Disk interface

Prism Business Systems

Prism House, 18-29 Mora Street, London EC1V 8BT
01 253 2277

Protek Computing Ltd

1A Young Square, Brucefield Industrial Pk, Livingston, W Lothian
0506 415353
Joystick interface

RD Laboratories

20 Court Road Estate, Cwmbran, Gwent NP44 3AS
06333 74333
Digital tracer

Ricoll Electronics Ltd

48 Southport Road, Ormskirk, Lancs L39 1QR
0695 79101
Keyboard

Saga Systems Ltd

Woodham Road, Woking, Surrey
Saga keyboards

Signpoint Ltd

Unit D, 166A Glyn Road, Clapton, London E5 0JE
01 533 0724
Sound unit

Sinclair Research

Stanhope Road, Camberley, Surrey GU15 3PS
0276 685311
Spectrum, Interface 1, Interface 2, Microdrive

Stack Computer Services Ltd

290-298 Derby Road, Bootle, Merseyside L20 8LN
051 933 5511
Light rifle

Stonechip Electronics Ltd

Unit 9, Brook Industrial Estate, Deadbrook Lane, Aldershot, Hants
0252 318260
Keyboard, joystick interface, BEEP amplifier

Success Services

154 High Street, Bloxwich, Walsall, W Midlands WS13 3JT
0922 402403
Pickard controller, joystick interface

Technology Research

356 Westmount Road, London SE9 1NW
0784 63547
Disk interface

Thurnall Electronics

95 Liverpool Road, Cadishead, Manchester M30 58G
061 775 4461
Joystick interface

Timedata

16 Hemmels, Laindon, Basildon, Essex SS15 6ED
0268 418121
ZXM sound and ZXS speech units

Transform

41 Keats House, Porchester Mead, Beckenham, Kent
01 658 6350
Keyboard

Trojan Products

166 Derlwyn Duvant, Swansea SA2 7PF
Lightpen

U-Microcomputers

Long Lane, Warrington, Cheshire
Expansion system

Voltmace Ltd

Park Drive, Baldock, Herts
0462 894410
Joysticks

William Stuart Systems

Quarley Down, Cholderton, Salisbury, Wiltshire
098064 235
Speech and sound systems

Zeal Marketing Ltd

Vanguard Trading Estate, Storforth Lane, Chesterfield S40 2TZ
0246 208555
BEEP amplifier

Appendix D

**697 Z-80 Operating Codes
with Mnemonics, Hex Codes,
Denary Codes and
Binary Codes**

Compiled by Ian Sinclair

The following list of all 697 Z-80 operating codes, with mnemonics, hex codes, denary codes, and binary codes consists of four columns. Where a code consists of more than one byte, the bytes have been arranged vertically, to avoid confusion. Where a data byte, displacement byte, or address has to be inserted, this is shown by using 00 for a data or displacement byte, or 0000 for an address. Customarily, this is shown on such lists by using N for a single byte or NN for an address (hence the position of these items in the otherwise alphabetical listing), but because a hex to denary and binary conversion program was used to produce the lists, letters such as N could not be used in the program.

The very considerable labour of producing such a list means that inevitably some mistakes are made during compiling the list. I have eliminated these as far as I know, but I shall be grateful if any errors are brought to my notice.

| | | | | | | | |
|----------------|----|-----|----------|----------------|----------------|--------------|--------------|
| ADC A, (HL) | 8E | 142 | 10001110 | AND (IY+00) | FD | 253 | 11111101 |
| ADC A, (IX+00) | DD | 221 | 11011101 | | A6 | 166 | 10100110 |
| | | | 8E | 142 | 10001110 | 00 | 0 00000000 |
| | | 00 | 0 | 00000000 | AND A | A7 | 167 10100111 |
| ADC A, (IY+00) | FD | 253 | 11111101 | AND B | A0 | 160 10100000 | |
| | | BE | 142 | 10001110 | AND C | A1 | 161 10100001 |
| | | 00 | 0 | 00000000 | AND D | A2 | 162 10100010 |
| ADC A,A | BF | 143 | 10001111 | AND 00 | E6 | 230 11100110 | |
| ADC A,B | 88 | 136 | 10001000 | | 00 | 0 00000000 | |
| ADC A,C | 89 | 137 | 10001001 | AND E | A3 | 163 10100011 | |
| ADC A,D | 8A | 138 | 10001010 | AND H | A4 | 164 10100100 | |
| ADC A,00 | CE | 206 | 11001110 | AND L | A5 | 165 10100101 | |
| | | 00 | 0 | 00000000 | BIT 0, (HL) | CB | 203 11001011 |
| ADC A,E | 8B | 139 | 10001011 | | 46 | 70 01000110 | |
| ADC A,H | 8C | 140 | 10001100 | BIT 0, (IX+00) | DD | 221 11011101 | |
| ADC A,L | 8D | 141 | 10001101 | | CB | 203 11001011 | |
| ADC HL,BC | ED | 237 | 11101101 | | 00 | 0 00000000 | |
| | | 4A | 74 | 01001010 | | 46 | 70 01000110 |
| ADC HL,DE | ED | 237 | 11101101 | BIT 0, (IY+00) | FD | 253 11111101 | |
| | | 5A | 90 | 01011010 | | CB | 203 11001011 |
| ADC HL,HL | ED | 237 | 11101101 | | 00 | 0 00000000 | |
| | | 6A | 106 | 01101010 | | 46 | 70 01000110 |
| ADC HL,SP | ED | 237 | 11101101 | BIT 0,A | CB | 203 11001011 | |
| | | 7A | 122 | 01111010 | | 47 | 71 01000111 |
| ADD A, (HL) | 86 | 134 | 10000110 | BIT 0,B | CB | 203 11001011 | |
| ADD A, (IX+00) | DD | 221 | 11011101 | | 40 | 64 01000000 | |
| | | 86 | 134 | 10000110 | BIT 0,C | CB | 203 11001011 |
| | | 00 | 0 | 00000000 | | 41 | 65 01000001 |
| ADD A, (IY+00) | FD | 253 | 11111101 | BIT 0,D | CB | 203 11001011 | |
| | | 86 | 134 | 10000110 | | 42 | 66 01000010 |
| | | 00 | 0 | 00000000 | BIT 0,E | CB | 203 11001011 |
| ADD A,A | 87 | 135 | 10000111 | | 43 | 67 01000011 | |
| ADD A,B | 80 | 128 | 10000000 | BIT 0,H | CB | 203 11001011 | |
| ADD A,C | 81 | 129 | 10000001 | | 44 | 68 01000100 | |
| ADD A,D | 82 | 130 | 10000010 | BIT 0,L | CB | 203 11001011 | |
| ADD A,00 | C6 | 198 | 11000110 | | 45 | 69 01000101 | |
| | | 00 | 0 | 00000000 | BIT 1, (HL) | CB | 203 11001011 |
| ADD A,E | 83 | 131 | 10000011 | | 4E | 78 01001110 | |
| ADD A,H | 84 | 132 | 10000100 | BIT 1, (IX+00) | DD | 221 11011101 | |
| ADD A,L | 85 | 133 | 10000101 | | CB | 203 11001011 | |
| ADD HL,BC | 09 | 9 | 00001001 | | 00 | 0 00000000 | |
| ADD HL,DE | 19 | 25 | 00011001 | | 4E | 78 01001110 | |
| ADD HL,HL | 29 | 41 | 00101001 | BIT 1, (IY+00) | FD | 253 11111101 | |
| ADD HL,SP | 39 | 57 | 00111001 | | CB | 203 11001011 | |
| ADD IX,BC | DD | 221 | 11011101 | | 00 | 0 00000000 | |
| | | 09 | 9 | 00001001 | | 4E | 78 01001110 |
| ADD IX,DE | DD | 221 | 11011101 | BIT 1,A | CB | 203 11001011 | |
| | | 19 | 25 | 00011001 | | 4F | 79 01001111 |
| ADD IX,IX | DD | 221 | 11011101 | BIT 1,B | CB | 203 11001011 | |
| | | 29 | 41 | 00101001 | | 48 | 72 01001000 |
| ADD IX,SP | DD | 221 | 11011101 | BIT 1,C | CB | 203 11001011 | |
| | | 39 | 57 | 00111001 | | 49 | 73 01001001 |
| ADD IY,BC | FD | 253 | 11111101 | BIT 1,D | CB | 203 11001011 | |
| | | 09 | 9 | 00001001 | | 4A | 74 01001010 |
| ADD IY,DE | FD | 253 | 11111101 | BIT 1,E | CB | 203 11001011 | |
| | | 19 | 25 | 00011001 | | 4B | 75 01001011 |
| ADD IY,IY | FD | 253 | 11111101 | BIT 1,H | CB | 203 11001011 | |
| | | 29 | 41 | 00101001 | | 4C | 76 01001100 |
| ADD IY,SP | FD | 253 | 11111101 | BIT 1,L | CB | 203 11001011 | |
| | | 39 | 57 | 00111001 | | 4D | 77 01001101 |
| AND (HL) | A6 | 166 | 10100110 | BIT 2, (HL) | CB | 203 11001011 | |
| AND (IX+00) | DD | 221 | 11011101 | | 56 | 86 01010110 | |
| | | A6 | 166 | 10100110 | BIT 2, (IX+00) | DD | 221 11011101 |
| | | 00 | 0 | 00000000 | | CB | 203 11001011 |
| | | 00 | 0 | 00000000 | | 56 | 86 01010110 |

| | | | | | | |
|----------------|----|-----|--------------|---------|----|--------------|
| BIT 2, (IY+00) | FD | 253 | 11111101 | BIT 4,L | CB | 203 11001011 |
| | | A6 | 166 10100110 | | CB | 203 11001011 |
| | | 00 | 0 00000000 | | 00 | 0 00000000 |
| | | 00 | 0 00000000 | | 56 | 86 01010110 |
| | | 00 | 0 00000000 | | 57 | 87 01010111 |
| | | 00 | 0 00000000 | | 58 | 88 01010000 |
| | | 00 | 0 00000000 | | 59 | 88 01010000 |
| | | 00 | 0 00000000 | | 51 | 81 01010111 |
| | | 00 | 0 00000000 | | 52 | 82 01010010 |
| | | 00 | 0 00000000 | | 53 | 83 01010011 |
| | | 00 | 0 00000000 | | 54 | 84 01010100 |
| | | 00 | 0 00000000 | | 55 | 85 01010101 |
| | | 00 | 0 00000000 | | 56 | 94 01011110 |
| | | 00 | 0 00000000 | | 57 | 95 01011111 |
| | | 00 | 0 00000000 | | 58 | 88 01011000 |
| | | 00 | 0 00000000 | | 59 | 89 01011001 |
| | | 00 | 0 00000000 | | 59 | 90 01011011 |
| | | 00 | 0 00000000 | | 60 | 91 01011101 |
| | | 00 | 0 00000000 | | 60 | 91 01011111 |
| | | 00 | 0 00000000 | | 61 | 92 01011100 |
| | | 00 | 0 00000000 | | 62 | 93 01011101 |
| | | 00 | 0 00000000 | | 63 | 92 01011110 |
| | | 00 | 0 00000000 | | 64 | 93 01011111 |
| | | 00 | 0 00000000 | | 65 | 91 01011111 |
| | | 00 | 0 00000000 | | 66 | 92 01011100 |
| | | 00 | 0 00000000 | | 67 | 93 01011101 |
| | | 00 | 0 00000000 | | 68 | 94 01011110 |
| | | 00 | 0 00000000 | | 69 | 95 01011111 |
| | | 00 | 0 00000000 | | 70 | 92 01110000 |
| | | 00 | 0 00000000 | | 71 | 112 01110000 |
| | | 00 | 0 00000000 | | 72 | 114 01110010 |
| | | 00 | 0 00000000 | | 73 | 115 01110011 |
| | | 00 | 0 00000000 | | 74 | 116 01110100 |
| | | 00 | 0 00000000 | | 75 | 117 01110101 |
| | | 00 | 0 00000000 | | 76 | 118 01110110 |
| | | 00 | 0 00000000 | | 77 | 119 01110111 |
| | | 00 | 0 00000000 | | 78 | 120 01110111 |
| | | 00 | 0 00000000 | | 79 | 121 01110111 |
| | | 00 | 0 00000000 | | 7E | 126 01111110 |
| | | 00 | 0 00000000 | | 7F | 127 01111111 |
| | | 00 | 0 00000000 | | 80 | 128 01111100 |

| | | | | | | | |
|------------|------------------|------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| BIT 7,C | CB 203 11001011 | DEC D | 15 21 00010101 | INC A | 3C 60 00111100 | LD (0000),A | 32 50 00110010 |
| | 79 121 01111001 | DEC DE | 1B 27 00011011 | INC B | 04 4 00000100 | | 00 0 00000000 |
| BIT 7,D | CB 203 11001011 | DEC E | 1D 29 00011101 | INC BC | 03 3 00000011 | | 00 0 00000000 |
| | 7A 122 01111010 | DEC H | 25 37 00100101 | INC C | 0C 12 00001100 | LD (0000),BC | ED 237 11101101 |
| BIT 7,E | CB 203 11001011 | DEC HL | 2B 43 00101011 | INC D | 14 20 00010100 | | 43 67 01000011 |
| | 7B 123 01111011 | DEC IX | DD 221 11011101 | INC DE | 13 19 00010011 | | 00 0 00000000 |
| BIT 7,H | CB 203 11001011 | | 2B 43 00101011 | INC E | 1C 28 00011100 | | 00 0 00000000 |
| | 7C 124 01111100 | DEC IY | FD 253 11111101 | INC H | 24 36 00100100 | LD (0000),DE | ED 237 11101101 |
| BIT 7,L | CB 203 11001011 | | 2B 43 00101011 | INC HL | 23 35 00100011 | | 53 83 01010011 |
| | 7D 125 01111101 | DEC L | 2D 45 00101101 | INC IX | DD 221 11011101 | | 00 0 00000000 |
| CALL 00 | CD 205 11001101 | DEC SP | 3B 59 00111011 | INC IY | 23 35 00100011 | | 00 0 00000000 |
| | 00 0 00000000 | DI | F3 243 11110011 | FD 253 11111101 | LD (0000),HL | ED 237 11101101 | |
| CALL C,00 | DC 220 11011100 | DJNZ,00 | 10 16 00010000 | | 23 35 00100011 | | 63 99 01100011 |
| | 00 0 00000000 | | 00 0 00000000 | INC L | 2C 44 00101100 | | 00 0 00000000 |
| CALL M,00 | FC 252 11111100 | EI | FB 251 11111011 | INC SP | 33 51 00110011 | | 00 0 00000000 |
| | 00 0 00000000 | EX | 8 8 10001000 | IND | ED 237 11101101 | LD (0000),HL | 22 34 00100010 |
| CALL NC,00 | D4 212 11010100 | EX(SP),HL | E3 227 11100011 | INDR | AA 170 10101010 | | 00 0 00000000 |
| | 00 0 00000000 | EX(SP)IX | DD 221 11011101 | ED 237 11101101 | | 00 0 00000000 | |
| CALL P,00 | F4 244 11110100 | | E3 227 11100011 | INI | BA 186 10111010 | LD (0000),IX | DD 221 11011101 |
| | 00 0 00000000 | EX(SP)IY | FD 253 11111101 | | ED 237 11101101 | | 22 34 00100010 |
| CALL PE,00 | EC 236 11101100 | | E3 227 11100011 | A2 162 10100010 | | 00 0 00000000 | |
| | 00 0 00000000 | EX AF,AF' | 08 8 00001000 | INIR | ED 237 11101101 | | 00 0 00000000 |
| CALL PO,00 | E4 228 11100100 | EX DE,HL | EB 235 11101011 | R2 178 10110010 | LD (0000),IY | FD 253 11111101 | |
| | 00 0 00000000 | EXX | D9 217 11011001 | JP(HL) | E9 233 11101001 | | 22 34 00100010 |
| CALL Z,00 | CC 204 11001100 | HLT | 76 118 01110110 | JP(IX) | DD 221 11011101 | | 00 0 00000000 |
| | 00 0 00000000 | IM 0 | ED 237 11101101 | E9 233 11101001 | | 00 0 00000000 | |
| CCF | 3F 63 00111111 | | 46 70 01000110 | JP(IY) | FD 253 11111101 | LD (0000),SP | ED 237 11101101 |
| CP(HL) | BE 190 10111110 | IM 1 | ED 237 11101101 | E9 233 11101001 | | 73 115 01110011 | |
| CP(IX+0) | DD 221 11011101 | | 56 86 01010110 | JP 0000 | C3 195 11000011 | | 00 0 00000000 |
| | BE 190 10111110 | IM 2 | ED 237 11101101 | 00 0 00000000 | | 00 0 00000000 | |
| | 00 0 00000000 | | 5E 94 01011110 | JP C,0000 | DA 218 11011010 | LD (BC),A | 02 2 00000010 |
| CP(IY+0) | FD 253 11111101 | IN A,(C) | ED 237 11101101 | | 00 0 00000000 | LD (DE),A | 12 18 00010010 |
| | BE 190 10111110 | | 78 120 01111000 | | 00 0 00000000 | LD (HL),A | 77 119 01110111 |
| | 00 0 00000000 | IN A,PORT | DB 219 11011011 | JP M,0000 | FA 250 11111010 | LD (HL),B | 70 112 01110000 |
| CP A | BF 191 10111111 | | 00 0 00000000 | | 00 0 00000000 | LD (HL),C | 71 113 01110001 |
| CP B | RB 184 10111100 | IN B,(C) | ED 237 11101101 | | 00 0 00000000 | LD (HL),D | 72 114 01110010 |
| CP C | RB 185 10111101 | | 40 64 01000000 | JP NC,0000 | D2 210 11010010 | LD (HL),E | 73 115 01110011 |
| CP D | BA 186 101111010 | IN C,(C) | ED 237 11101101 | | 00 0 00000000 | LD (HL),H | 74 116 01110100 |
| CP 00 | FE 254 11111110 | | 48 72 01001000 | JP NZ,0000 | C2 194 11000010 | LD (HL),L | 75 117 01110101 |
| | 00 0 00000000 | IN D,(C) | ED 237 11101101 | | 00 0 00000000 | LD (IX+00),A | DD 221 11011101 |
| CP E | RB 187 10111011 | | 50 80 01010000 | | 00 0 00000000 | | 00 0 00000000 |
| CP H | BC 188 10111100 | IN E,(C) | ED 237 11101101 | JP P,0000 | F2 242 11110010 | LD (HL),E | 73 115 01110011 |
| CP L | BD 189 10111101 | | 58 88 01010000 | | 00 0 00000000 | LD (HL),H | 74 116 01110100 |
| CPD | ED 237 11101101 | IN H,(C) | ED 237 11101101 | JP NZ,0000 | C2 194 11000010 | LD (HL),L | 75 117 01110101 |
| | A9 169 10101001 | | 60 96 01100000 | | 00 0 00000000 | LD (HL),L | 77 119 01110111 |
| CPDR | ED 237 11101101 | IN L,(C) | ED 237 11101101 | JP PE,0000 | EA 234 11101010 | LD (HL),L | 77 119 01110111 |
| | B9 185 10111001 | | 68 104 01101000 | | 00 0 00000000 | LD (IX+00),C | DD 221 11011101 |
| CPI | ED 237 11101101 | INC(HL) | 34 52 00110100 | | 00 0 00000000 | DD 221 11011101 | |
| | A1 161 10100001 | INC(IX+00) | DD 221 11011101 | JP PO,0000 | E2 226 11100010 | 71 113 01110001 | |
| CPIR | ED 237 11101101 | | 34 52 00110100 | | 00 0 00000000 | 00 0 00000000 | |
| | B1 177 10110001 | | 00 0 00000000 | JP PO,0000 | E2 226 11100010 | LD (IX+00),B | DD 221 11011101 |
| CPL | 2F 47 00101111 | INC(IY+00) | DD 221 11011101 | | 00 0 00000000 | | 00 0 00000000 |
| DAA | 27 39 00100111 | | 34 52 00110100 | JP Z,0000 | CA 202 11001010 | DD 221 11011101 | |
| DEC(HL) | 35 53 00110101 | | 00 0 00000000 | | 00 0 00000000 | | 00 0 00000000 |
| DEC(IX+0) | DD 221 11011101 | INC A | 3C 60 00111100 | JP Z,0000 | CA 202 11001010 | | 00 0 00000000 |
| | 35 53 00110101 | INC B | 04 4 00000100 | | 00 0 00000000 | | 00 0 00000000 |
| | 00 0 00000000 | INC BC | 03 3 00000011 | JR C,00 | 38 56 00111000 | | 00 0 00000000 |
| DEC(IY+0) | FD 253 11111101 | INC C | 0C 12 00001100 | | 00 0 00000000 | | 00 0 00000000 |
| | 35 53 00110101 | INC D | 14 20 00010100 | JR 00 | 18 24 00011000 | LD (IX+00),E | DD 221 11011101 |
| | 00 0 00000000 | INC DE | 13 19 00010011 | | 00 0 00000000 | | 00 0 00000000 |
| DEC A | 3D 61 00111101 | INC E | 1C 28 00011100 | JR NC,00 | 30 48 00110000 | LD (IX+00),E | DD 221 11011101 |
| DEC B | 05 5 00000101 | INC H | 24 36 00100100 | | 00 0 00000000 | | 00 0 00000000 |
| DEC BC | 0B 11 00001011 | INC HL | 23 35 00100011 | JR NZ,00 | 20 32 00100000 | LD (IX+00),H | DD 221 11011101 |
| DEC C | 0D 13 00001101 | INC IX | DD 221 11011101 | | 00 0 00000000 | | 00 0 00000000 |
| | | | 23 35 00100011 | JR Z,00 | 28 40 00101000 | LD (IX+00),L | DD 221 11011101 |
| | | | | | 00 0 00000000 | | 00 0 00000000 |
| | | | | | 00 0 00000000 | | 00 0 00000000 |

| | | | | | | | | | | | | | | | | | |
|--------------|----|-----|----------|--------------|--------------|-----|----------|--------------|--------------|-----|----------|----------|------------|----------|----------|----------|----------|
| LD(IY+00),E | FD | 253 | 11111101 | LD C,A | 4F | 79 | 01001111 | | LD H,(HL) | 66 | 102 | 01100110 | LD SP,0000 | 31 | 49 | 00110001 | |
| | | 73 | 115 | 01110011 | LD C,B | 4B | 72 | 01001000 | LD H,(IX+00) | DD | 221 | 11011101 | | 00 | 0 | 00000000 | |
| | | 00 | 0 | 00000000 | LD C,C | 49 | 73 | 01001001 | | 66 | 102 | 01100110 | | 00 | 0 | 00000000 | |
| LD(IY+00),H | FD | 253 | 11111101 | LD C,D | 4A | 74 | 01001010 | | | 00 | 0 | 00000000 | LD SP,HL | F9 | 249 | 11111001 | |
| | | 74 | 116 | 01110100 | LD C,00 | 0E | 14 | 00001110 | LD H,(IY+00) | FD | 253 | 11111101 | LD SP,IX | DD | 221 | 11011101 | |
| | | 00 | 0 | 00000000 | | 00 | 0 | 00000000 | | 66 | 102 | 01100110 | | F9 | 249 | 11111001 | |
| LD(IY+00),L | FD | 253 | 11111101 | LD C,E | 4B | 75 | 01001011 | | LD H,A | 67 | 103 | 01100111 | LD SP,1Y | FD | 253 | 11111101 | |
| | | 75 | 117 | 01110101 | LD C,H | 4C | 76 | 01001100 | LD H,B | 60 | 96 | 01100000 | | F9 | 249 | 11111001 | |
| | | 00 | 0 | 00000000 | LD C,L | 4D | 77 | 01001101 | LD H,C | 61 | 97 | 01100001 | LDD | ED | 237 | 11011101 | |
| LD A,(0000) | 3A | 58 | 00111010 | LD D,(HL) | 56 | 86 | 01010110 | LD H,D | 62 | 98 | 01100010 | LDDR | ED | 237 | 11011101 | | |
| | | 00 | 0 | 00000000 | LD D,(IX+00) | DD | 221 | 11011101 | LD H,00 | 26 | 38 | 00100110 | | BB | 184 | 10111000 | |
| | | 00 | 0 | 00000000 | | 56 | 86 | 01010110 | | 00 | 0 | 00000000 | LDI | ED | 237 | 11011101 | |
| LD A,(BC) | 0A | 10 | 00001010 | | 00 | 0 | 00000000 | | LD H,E | 63 | 99 | 01100011 | | AO | 160 | 10100000 | |
| LD A,(DE) | 1A | 26 | 00011010 | LD D,(IY+00) | FD | 253 | 11111101 | LD H,H | 64 | 100 | 01100100 | LDIIR | ED | 237 | 11011101 | | |
| LD A,(HL) | 7E | 126 | 01111110 | | 56 | 86 | 01010110 | LD H,L | 65 | 101 | 01100101 | | BO | 176 | 10110000 | | |
| LD A,(IY+00) | DD | 221 | 11011101 | | 00 | 0 | 00000000 | LD HL,(0000) | ED | 237 | 11011101 | NEG | ED | 237 | 11011101 | | |
| | | 7E | 126 | 01111110 | LD D,A | 57 | 87 | 01010111 | | 68 | 107 | 01101011 | | 44 | 68 | 01000100 | |
| | | 00 | 0 | 00000000 | LD D,B | 50 | 80 | 01010000 | LD HL,(0000) | | 00 | 0 | 00000000 | NOP | 00 | 0 | 00000000 |
| LD A,(IX+00) | FD | 253 | 11111101 | LD D,C | 51 | 81 | 01010001 | | 00 | 0 | 00000000 | OR(HL) | B6 | 182 | 10110110 | | |
| | | 7E | 126 | 01111110 | LD D,D | 52 | 82 | 01010010 | LD HL,(0000) | 2A | 42 | 00101010 | OR(IX+00) | DD | 221 | 11011101 | |
| | | 00 | 0 | 00000000 | LD D,00 | 16 | 22 | 00010110 | | 00 | 0 | 00000000 | | B6 | 182 | 10110110 | |
| LD A,A | 7F | 127 | 01111111 | | 00 | 0 | 00000000 | | LD HL,0000 | 21 | 33 | 00100001 | | | 00 | 0 | 00000000 |
| LD A,B | 78 | 120 | 01111000 | LD D,E | 53 | 83 | 01010011 | | | 00 | 0 | 00000000 | OR(IY+00) | FD | 253 | 11111101 | |
| LD A,C | 79 | 121 | 01111001 | LD D,H | 54 | 84 | 01010100 | | | 00 | 0 | 00000000 | | B6 | 182 | 10110110 | |
| LD A,D | 7A | 122 | 01111010 | LD D,L | 55 | 85 | 01010101 | | LD I,A | ED | 237 | 11011101 | | | 00 | 0 | 00000000 |
| LD A,00 | 3E | 62 | 00111110 | LD DE,(0000) | ED | 237 | 11011101 | | | 47 | 71 | 01000111 | OR A | B7 | 183 | 10110111 | |
| | | 00 | 0 | 00000000 | | 5B | 91 | 01011011 | LD IX,(0000) | DD | 221 | 11011101 | OR B | BO | 176 | 10110000 | |
| LD A,E | 7B | 123 | 01111011 | | 00 | 0 | 00000000 | | | 2A | 42 | 00101010 | OR C | BI | 177 | 10110001 | |
| LD A,H | 7C | 124 | 01111100 | | 00 | 0 | 00000000 | | | 00 | 0 | 00000000 | OR D | B2 | 178 | 10110010 | |
| LD A,I | ED | 237 | 11011101 | LD DE,0000 | 11 | 17 | 00010001 | | | 00 | 0 | 00000000 | OR 00 | F6 | 246 | 11110110 | |
| | | 57 | 87 | 01010111 | | 00 | 0 | 00000000 | LD IX,0000 | DD | 221 | 11011101 | | | 00 | 0 | 00000000 |
| LD A,L | 7D | 125 | 01111101 | | 00 | 0 | 00000000 | | | 21 | 33 | 00100001 | OR E | BS | 179 | 10110011 | |
| LD A,R | ED | 237 | 11011101 | LD E,(HL) | 5E | 94 | 01011110 | | | 00 | 0 | 00000000 | OR H | BS | 180 | 10110100 | |
| | | SF | 95 | 01011111 | LD E,(IX+00) | DD | 221 | 11011101 | | | 00 | 0 | 00000000 | OR L | BS | 181 | 10110101 |
| LD B,(HL) | 46 | 70 | 01000110 | | 5E | 94 | 01011110 | | LD IY,(0000) | FD | 253 | 11111101 | OTDR | ED | 237 | 11011101 | |
| LD B,(IX+00) | DD | 221 | 11011101 | | 00 | 0 | 00000000 | | | 2A | 42 | 00101010 | | BB | 187 | 10111011 | |
| | | 46 | 70 | 01000110 | LD E,(IY+00) | FD | 253 | 11111101 | | | 00 | 0 | 00000000 | OTIR | ED | 237 | 11011101 |
| | | 00 | 0 | 00000000 | | 5E | 94 | 01011110 | | | 00 | 0 | 00000000 | | B3 | 179 | 10110011 |
| LD B,(IY+00) | FD | 253 | 11111101 | | 00 | 0 | 00000000 | LD IY,0000 | FD | 253 | 11111101 | | | 00 | 0 | 00000000 | |
| | | 46 | 70 | 01000110 | LD E,A | 5F | 95 | 01011111 | | 21 | 33 | 00100001 | OUT(C),A | ED | 237 | 11011101 | |
| | | 00 | 0 | 00000000 | LD E,B | 5B | 88 | 01011000 | | | 00 | 0 | 00000000 | OUT(C),B | 79 | 121 | 01111001 |
| LD B,A | 47 | 71 | 01000111 | LD E,C | 59 | 89 | 01011001 | | | 00 | 0 | 00000000 | | | ED | 237 | 11011101 |
| LD B,B | 40 | 64 | 01000000 | LD E,D | 5A | 90 | 01011010 | | LD L,(HL) | 6E | 110 | 01101110 | OUT(C),C | 41 | 65 | 01000001 | |
| LD B,C | 41 | 65 | 01000001 | LD E,00 | 1E | 30 | 00011110 | | LD L,(IX+00) | DD | 221 | 11011101 | | | 49 | 73 | 01001001 |
| LD B,D | 42 | 66 | 01000010 | | 00 | 0 | 00000000 | | | 6E | 110 | 01101110 | OUT(C),D | ED | 237 | 11011101 | |
| LD B,00 | 06 | 6 | 00000110 | LD E,E | 5B | 91 | 01011011 | | | 00 | 0 | 00000000 | | S1 | 81 | 01010001 | |
| | | 00 | 0 | 00000000 | LD E,H | 5C | 92 | 01011100 | LD L,B | 68 | 104 | 01101000 | | | ED | 237 | 11011101 |
| LD B,E | 43 | 67 | 01000011 | LD E,L | 5D | 93 | 01011101 | | LD L,C | 69 | 105 | 01101001 | OUT(C),L | 69 | 105 | 01101001 | |
| LD B,H | 44 | 68 | 01000100 | LD H,(HL) | 66 | 102 | 01100110 | | LD L,D | 6A | 106 | 01101010 | OUT00,A | D3 | 211 | 11010011 | |
| LD B,L | 45 | 69 | 01000101 | LD H,(IX+00) | DD | 221 | 11011101 | | LD L,00 | 2E | 46 | 00101110 | | | 61 | 97 | 01100001 |
| LD BC,(0000) | ED | 237 | 11011101 | | 66 | 102 | 01100110 | | | 00 | 0 | 00000000 | OUTD | ED | 237 | 11011101 | |
| | | 4B | 75 | 01001011 | | 00 | 0 | 00000000 | LD L,E | 6B | 107 | 01101011 | | AB | 171 | 10101011 | |
| | | 00 | 0 | 00000000 | LD H,(IY+00) | FD | 253 | 11111101 | LD L,H | 6C | 108 | 01101100 | OUTI | ED | 237 | 11011101 | |
| | | 00 | 0 | 00000000 | | 66 | 102 | 01100110 | LD L,L | 6D | 109 | 01101101 | | A3 | 163 | 10100011 | |
| LD BC,0000 | 01 | 1 | 00000001 | | 00 | 0 | 00000000 | LD R,A | ED | 237 | 11011101 | POP AF | F1 | 241 | 11110001 | | |
| | | 00 | 0 | 00000000 | LD H,A | 67 | 103 | 01100111 | | 4F | 79 | 01001111 | POP BC | C1 | 193 | 11000001 | |
| | | 00 | 0 | 00000000 | LD H,B | 60 | 96 | 01100000 | LD SP,(0000) | ED | 237 | 11011101 | POP DE | D1 | 209 | 11010001 | |
| LD C,(HL) | 4E | 78 | 01001110 | LD H,C | 61 | 97 | 01100001 | | 7B | 123 | 01111011 | POP HL | E1 | 225 | 11000001 | | |
| LD C,(IX+00) | DD | 221 | 11011101 | LD H,D | 62 | 98 | 01100010 | | | 00 | 0 | 00000000 | POP IX | DD | 221 | 11011101 | |
| | | 4E | 78 | 01001110 | LD H,00 | 26 | 38 | 00100110 | | 00 | 0 | 00000000 | | E1 | 225 | 11000001 | |
| LD C,(IY+00) | FD | 253 | 11111101 | LD H,E | 63 | 99 | 01100011 | | | 00 | 0 | 00000000 | | | | | |
| | | 4E | 78 | 01001110 | LD H,H | 64 | 100 | 01100100 | | | 00 | 0 | 00000000 | | | | |
| | | 00 | 0 | 00000000 | LD H,L | 65 | 101 | 01100101 | | | 00 | 0 | 00000000 | | | | |

| | | | | | | | | | | | | | | | |
|----------------|----|-----|----------|---------------|----|-----|----------|---------------|----|-----|----------|------------|-----|----------|----------|
| RES 0, (1Y+00) | FD | 253 | 11111101 | RES 2,L | CB | 203 | 11001011 | RES 4,L | CB | 203 | 11001011 | RES 7,C | CB | 203 | 11001011 |
| | CB | 203 | 11001011 | | 95 | 149 | 10010101 | | AS | 165 | 10100101 | | B9 | 185 | 10111001 |
| | 00 | 0 | 00000000 | RES 3,(HL) | CB | 203 | 11001011 | RES 5,(HL) | CB | 203 | 11001011 | RES 7,D | CB | 203 | 11001011 |
| | 00 | 0 | 00000000 | | 9E | 158 | 10011110 | | AE | 174 | 10101110 | | BA | 186 | 10111010 |
| RES 0,A | CB | 203 | 11001011 | RES 3,(IX+00) | DD | 221 | 11011101 | RES 5,(IX+00) | DD | 221 | 11011101 | RES 7,E | CB | 203 | 11001011 |
| | 87 | 135 | 10000111 | | CB | 203 | 11001011 | | CB | 203 | 11001011 | | BB | 187 | 10111011 |
| RES 0,B | CB | 203 | 11001011 | | 00 | 0 | 00000000 | | 00 | 0 | 00000000 | RES 7,H | CB | 203 | 11001011 |
| | 80 | 128 | 10000000 | | 9E | 158 | 10011110 | | AE | 174 | 10101110 | | BC | 188 | 10111100 |
| RES 0,C | CB | 203 | 11001011 | RES 3,(IY+00) | FD | 253 | 11111101 | RES 5,(IY+00) | FD | 253 | 11111101 | RES 7,L | CB | 203 | 11001011 |
| | 81 | 129 | 10000001 | | CB | 203 | 11001011 | | CB | 203 | 11001011 | | BD | 189 | 10111101 |
| RES 0,D | CB | 203 | 11001011 | | 00 | 0 | 00000000 | | 00 | 0 | 00000000 | RET | CB | 201 | 11001001 |
| | 82 | 130 | 10000010 | | 9E | 158 | 10011110 | | AE | 174 | 10101110 | RET C | DB | 216 | 11011000 |
| RES 0,E | CB | 203 | 11001011 | RES 3,A | CB | 203 | 11001011 | RES 5,A | CB | 203 | 11001011 | RET M | FB | 248 | 11111000 |
| | 83 | 131 | 10000011 | | 9F | 159 | 10011111 | | AF | 175 | 10101111 | RET NC | DO | 208 | 11010000 |
| RES 0,H | CB | 203 | 11001011 | RES 3,B | CB | 203 | 11001011 | RES 5,B | CB | 203 | 11001011 | RET NZ | CO | 192 | 11000000 |
| | 84 | 132 | 10000100 | | 98 | 152 | 10011000 | | AB | 168 | 10101000 | RET P | FO | 240 | 11110000 |
| RES 0,L | CB | 203 | 11001011 | RES 3,C | CB | 203 | 11001011 | RES 5,C | CB | 203 | 11001011 | RET PE | EB | 232 | 11101000 |
| | 85 | 133 | 10000101 | | 99 | 153 | 10011001 | | A9 | 169 | 10101001 | RET PO | EO | 224 | 11100000 |
| RES 1,(HL) | CB | 203 | 11001011 | RES 3,D | CB | 203 | 11001011 | RES 5,D | CB | 203 | 11001011 | RET Z | CB | 200 | 11001000 |
| | 8E | 142 | 10001110 | | 9A | 154 | 10011010 | | AA | 170 | 10101010 | RETI | ED | 237 | 11101101 |
| RES 1,(IX+00) | DD | 221 | 11011101 | RES 3,E | CB | 203 | 11001011 | RES 5,E | CB | 203 | 11001011 | 4D | 77 | 01001101 | |
| | CB | 203 | 11001011 | | 9B | 155 | 10011011 | | AB | 171 | 10101011 | RETN | ED | 237 | 11101101 |
| | 00 | 0 | 00000000 | RES 3,H | CB | 203 | 11001011 | RES 5,HCBAC | CB | 203 | 11001011 | 45 | 69 | 01000101 | |
| | 00 | 0 | 00000000 | | 9C | 156 | 10011100 | | AC | 172 | 10101100 | RL(HL) | CB | 203 | 11001011 |
| RES 1,(IY+00) | FD | 253 | 11111101 | RES 3,L | CB | 203 | 11001011 | RES 5,L | CB | 203 | 11001011 | 16 | 22 | 00010110 | |
| | CB | 203 | 11001011 | | 9D | 157 | 10011101 | | AD | 173 | 10101101 | RL(IX+00) | DD | 221 | 11011101 |
| | 00 | 0 | 00000000 | RES 4,(HL) | CB | 203 | 11001011 | RES 6,(HL) | CB | 203 | 11001011 | CB | 203 | 11001011 | |
| | 00 | 0 | 00000000 | | A6 | 166 | 10100110 | | B6 | 182 | 10110110 | 00 | 0 | 00000000 | |
| RES 1,A | CB | 203 | 11001011 | RES 4,(IX+00) | DD | 221 | 11011101 | RES 6,(IX+00) | DD | 221 | 11011101 | 16 | 22 | 00010110 | |
| | 8F | 143 | 10001111 | | CB | 203 | 11001011 | | CB | 203 | 11001011 | RL(IY+00) | FD | 253 | 11111101 |
| RES 1,B | CB | 203 | 11001011 | | 00 | 0 | 00000000 | | 00 | 0 | 00000000 | | CB | 203 | 11001011 |
| | 88 | 136 | 10001000 | | A6 | 166 | 10100110 | | B6 | 182 | 10110110 | | 00 | 0 | 00000000 |
| RES 1,C | CB | 203 | 11001011 | RES 4,(IY+00) | FD | 253 | 11111101 | RES 6,(IY+00) | FD | 253 | 11111101 | 16 | 22 | 00010110 | |
| | 89 | 137 | 10001001 | | CB | 203 | 11001011 | | CB | 203 | 11001011 | RL A | CB | 203 | 11001011 |
| RES 1,D | CB | 203 | 11001011 | | 00 | 0 | 00000000 | | 00 | 0 | 00000000 | | 17 | 23 | 00010111 |
| | 8A | 138 | 10001010 | | A6 | 166 | 10100110 | | B6 | 182 | 10110110 | RL B | CB | 203 | 11001011 |
| RES 1,E | CB | 203 | 11001011 | RES 4,A | CB | 203 | 11001011 | RES 6,A | CB | 203 | 11001011 | 18 | 16 | 00010000 | |
| | 8B | 139 | 10001011 | | A7 | 167 | 10100111 | | B7 | 183 | 10110111 | RL C | CB | 203 | 11001011 |
| RES 1,H | CB | 203 | 11001011 | RES 4,B | CB | 203 | 11001011 | RES 6,B | CB | 203 | 11001011 | 19 | 17 | 00010001 | |
| | 8C | 140 | 10001100 | | A0 | 160 | 10100000 | | B0 | 176 | 10110000 | RL D | CB | 203 | 11001011 |
| RES 1,L | CB | 203 | 11001011 | RES 4,C | CB | 203 | 11001011 | RES 6,C | CB | 203 | 11001011 | 12 | 18 | 00010010 | |
| | 8D | 141 | 10001101 | | A1 | 161 | 10100001 | | B1 | 177 | 10110001 | RL E | CB | 203 | 11001011 |
| RES 2,(HL) | CB | 203 | 11001011 | RES 4,D | CB | 203 | 11001011 | RES 6,D | CB | 203 | 11001011 | 13 | 19 | 00010011 | |
| | 96 | 150 | 10010110 | | A2 | 162 | 10100010 | | B2 | 178 | 10110010 | RL H | CB | 203 | 11001011 |
| RES 2,(IX+00) | DD | 221 | 11011101 | RES 4,E | CB | 203 | 11001011 | RES 6,E | CB | 203 | 11001011 | 14 | 20 | 00010100 | |
| | CB | 203 | 11001011 | | A3 | 163 | 10100011 | | B3 | 179 | 10110011 | RL L | CB | 203 | 11001011 |
| | 00 | 0 | 00000000 | RES 4,H | CB | 203 | 11001011 | RES 6,H | CB | 203 | 11001011 | 15 | 21 | 00010101 | |
| | 96 | 150 | 10010110 | | A4 | 164 | 10100100 | | B4 | 180 | 10110100 | RLA | CB | 203 | 11001011 |
| RES 2,(IY+00) | FD | 253 | 11111101 | RES 4,L | CB | 203 | 11001011 | RES 6,L | CB | 203 | 11001011 | 16 | 6 | 00000110 | |
| | CB | 203 | 11001011 | | A5 | 165 | 10100101 | | BE | 190 | 10110101 | RLC(HL) | CB | 203 | 11001011 |
| | 00 | 0 | 00000000 | RES 5,(HL) | CB | 203 | 11001011 | RES 7,(HL) | CB | 203 | 11001011 | 06 | 6 | 00000110 | |
| | 96 | 150 | 10010110 | | AE | 174 | 10101110 | | BE | 190 | 10111110 | RLC(IX+00) | DD | 221 | 11011101 |
| RES 2,A | CB | 203 | 11001011 | RES 5,(IX+00) | DD | 221 | 11011101 | | CB | 203 | 11001011 | 06 | 6 | 00000110 | |
| | 97 | 151 | 10010111 | | CB | 203 | 11001011 | | 00 | 0 | 00000000 | RLC(IY+00) | FD | 253 | 11111101 |
| RES 2,B | CB | 203 | 11001011 | | 00 | 0 | 00000000 | | BE | 190 | 10111110 | | CB | 203 | 11001011 |
| | 90 | 144 | 10010000 | | AE | 174 | 10101110 | RES 7,(IX+00) | DD | 221 | 11011101 | 06 | 6 | 00000110 | |
| RES 2,C | CB | 203 | 11001011 | RES 5,(IY+00) | FD | 253 | 11111101 | | CB | 203 | 11001011 | 06 | 6 | 00000110 | |
| | 91 | 145 | 10010001 | | CB | 203 | 11001011 | | 00 | 0 | 00000000 | RLC(IY+00) | FD | 253 | 11111101 |
| RES 2,D | CB | 203 | 11001011 | | 00 | 0 | 00000000 | | BE | 190 | 10111110 | | CB | 203 | 11001011 |
| | 92 | 146 | 10010010 | | AE | 174 | 10101110 | RES 7,(IY+00) | FD | 253 | 11111101 | 06 | 6 | 00000110 | |
| RES 2,E | CB | 203 | 11001011 | RES 5,A | CB | 203 | 11001011 | | CB | 203 | 11001011 | 06 | 6 | 00000110 | |
| | 93 | 147 | 10010011 | | 5F | 95 | 01011111 | | 00 | 0 | 00000000 | RLC A | CB | 203 | 11001011 |
| RES 2,H | CB | 203 | 11001011 | RES 5,B | CB | 203 | 11001011 | RES 7,A | CB | 203 | 11001011 | 07 | 7 | 00000111 | |
| | 94 | 148 | 10010100 | | AB | 168 | 10101000 | | BE | 191 | 10111111 | RLC B | CB | 203 | 11001011 |
| | | | | | | | | RES 7,B | CB | 203 | 11001011 | 06 | 6 | 00000000 | |
| | | | | | | | | | BB | 184 | 10111000 | RLC C | CB | 203 | 11001011 |
| | | | | | | | | | | 01 | 1 | 00000001 | | | |

| | | | | | | | | | | | | |
|----------------|--------|----------|---------------|--------|----------|---------------|--------|----------|---------------|---------|----------|----------|
| RR (IY+00) | FD 253 | 11111101 | SBC A, D | 9A 154 | 10011010 | SET1, A | CB 203 | 11001011 | SET4, (IX+00) | DD 221 | 11011101 | |
| | CB 203 | 11001011 | SBC A, 00 | DE 222 | 11011110 | SET1, B | CF 207 | 11001111 | | CB 203 | 11001011 | |
| | 00 0 | 00000000 | | 00 0 | 00000000 | | CB 203 | 11001011 | | 00 0 | 00000000 | |
| | 1E 30 | 00011110 | SBC A, E | 9B 155 | 10011011 | SET1, C | CB 203 | 11001000 | | E6 230 | 11100110 | |
| RR A | CB 203 | 11001011 | SBC A, H | 9C 156 | 10011100 | | CB 203 | 11001011 | SET4, (IY+00) | FD 253 | 11111101 | |
| | 1F 31 | 00011111 | SBC A, L | 9D 157 | 10011101 | SET1, D | CB 203 | 11001011 | | CB 203 | 11001011 | |
| RR B | CB 203 | 11001011 | SBC HL, BC | ED 237 | 11101101 | | CA 202 | 11001010 | | 00 0 | 00000000 | |
| | 18 24 | 00011000 | | 42 66 | 01000010 | SET1, E | CB 203 | 11001011 | | E6 230 | 11100110 | |
| RR C | CB 203 | 11001011 | SBC HL, DE | ED 237 | 11101101 | | CB 203 | 11001011 | SET4, A | CB 203 | 11001011 | |
| | 19 25 | 00011001 | | 52 82 | 01010010 | SET1, F | CB 203 | 11001011 | | E7 231 | 11100111 | |
| RR D | CB 203 | 11001011 | SBC HL, HL | ED 237 | 11101101 | SET1, G | CB 203 | 11001011 | SET4, B | CB 203 | 11001011 | |
| | 1A 26 | 00011010 | | 62 98 | 01100010 | | CC 204 | 11001100 | | E0 224 | 11100000 | |
| RR E | CB 203 | 11001011 | SBC HL, SP | ED 237 | 11101101 | SET1, H | CB 203 | 11001011 | SET4, C | CB 203 | 11001011 | |
| | 1B 27 | 00011011 | | 72 114 | 01110010 | | CD 205 | 11001101 | | E1 225 | 11100001 | |
| RR H | CB 203 | 11001011 | SCF | 37 55 | 00110111 | SET2, (HL) | CB 203 | 11001011 | SET4, D | CB 203 | 11001011 | |
| | 1C 28 | 00011100 | SET 0, (HL) | CB 203 | 11001011 | | D6 214 | 11010110 | | E2 226 | 11100010 | |
| RR L | CB 203 | 11001011 | | C6 198 | 11000110 | SET2, (IX+00) | DD 221 | 11011101 | SET4, E | CB 203 | 11001011 | |
| | 1D 29 | 00011101 | SET0, (IX+00) | DD 221 | 11011101 | | CB 203 | 11001011 | | E3 227 | 11100011 | |
| RRA | 1F 31 | 00011111 | | CB 203 | 11001011 | | 00 0 | 00000000 | SET4, F | CB 203 | 11001011 | |
| RRC (HL) | CB 203 | 11001011 | | 00 0 | 00000000 | | D6 214 | 11010110 | | E4 228 | 11100100 | |
| | OE 14 | 00001110 | | C6 198 | 11000110 | SET2, (IY+00) | FD 253 | 11111101 | SET4, G | CB 203 | 11001011 | |
| RRC (IX+00) | DD 221 | 11011101 | SET0, (IY+00) | FD 253 | 11111101 | | CB 203 | 11001011 | | ES 229 | 11100101 | |
| | CB 203 | 11001011 | | CB 203 | 11001011 | | 00 0 | 00000000 | SET5, (HL) | CB 203 | 11001011 | |
| | 00 0 | 00000000 | | 00 0 | 00000000 | | D6 214 | 11010110 | | EE 238 | 11101110 | |
| | OE 14 | 00001110 | | 23 35 | 00100011 | SET2, A | CB 203 | 11001011 | SET5, (IX+00) | DD 221 | 11011101 | |
| RRC (IY+00) | FD 253 | 11111101 | SET0, A | CB 203 | 11001011 | | D7 215 | 11010111 | | CB 203 | 11001011 | |
| | CB 203 | 11001011 | | C7 199 | 11000111 | SET2, B | CB 203 | 11001011 | | 00 0 | 00000000 | |
| | 00 0 | 00000000 | SET0, B | CB 203 | 11001011 | | D0 208 | 11010000 | | EE 238 | 11101110 | |
| | OE 14 | 00001110 | | CO 192 | 11000000 | SET2, C | CB 203 | 11001011 | SET5, (IY+00) | FD 253 | 11111101 | |
| RRC A | CB 203 | 11001011 | SET0, C | CB 203 | 11001011 | | D1 209 | 11010001 | | CB 203 | 11001011 | |
| | OF 15 | 00001111 | | C1 193 | 11000001 | SET2, D | CB 203 | 11001011 | | 00 0 | 00000000 | |
| RRC B | CB 203 | 11001011 | SET0, D | CB 203 | 11001011 | SET2, E | CB 203 | 11001011 | SET5, A | CB 203 | 11001011 | |
| | CB 203 | 11001011 | | C2 194 | 11000010 | | D3 211 | 11010011 | | EF 239 | 11101111 | |
| RRC C | CB 203 | 11001011 | SET0, E | CB 203 | 11001011 | SET2, F | CB 203 | 11001011 | SET5, B | CB 203 | 11001011 | |
| | 09 9 | 00001001 | | C3 195 | 11000011 | | D4 212 | 11010100 | | E8 232 | 11101000 | |
| RRC D | CB 203 | 11001011 | SET0, H | CB 203 | 11001011 | SET2, G | CB 203 | 11001011 | SET5, C | CB 203 | 11001011 | |
| | 0A 10 | 00001010 | | C4 196 | 11000100 | | D5 213 | 11010101 | | E9 233 | 11101001 | |
| RRC E | CB 203 | 11001011 | SET0, L | CB 203 | 11001011 | SET3, (HL) | CB 203 | 11001011 | SET5, D | CB 203 | 11001011 | |
| | 0B 11 | 00001011 | | C5 197 | 11000101 | | DE 222 | 11011110 | | EA 234 | 11101010 | |
| RRC H | CB 203 | 11001011 | SET1, (HL) | CB 203 | 11001011 | SET3, (IX+00) | DD 221 | 11011101 | SET5, E | CB 203 | 11001011 | |
| | 0C 12 | 00001100 | | CE 206 | 11001110 | | CB 203 | 11001011 | | EB 235 | 11101011 | |
| RRC L | CB 203 | 11001011 | SET1, (IX+00) | DD 221 | 11011101 | | 00 0 | 00000000 | SET5, F | CB 203 | 11001011 | |
| | 0D 13 | 00001101 | | CB 203 | 11001011 | | DE 222 | 11011110 | | EC 236 | 11101100 | |
| RRCA | OF 15 | 00001111 | | 00 0 | 00000000 | SET3, (IY+00) | FD 253 | 11111101 | SET5, G | CB 203 | 11001011 | |
| RRD | ED 237 | 11101101 | | CE 206 | 11001110 | | CB 203 | 11001011 | | ED 237 | 11101101 | |
| | 67 103 | 01100111 | SET1, (IY+00) | FD 253 | 11111101 | | 00 0 | 00000000 | SET6, (HL) | CB 203 | 11001011 | |
| RST 00 | C7 199 | 11000111 | | CB 203 | 11001011 | | DE 222 | 11011110 | | F6 246 | 11101110 | |
| RST 08 | CF 207 | 11001111 | | 00 0 | 00000000 | SET3, A | CB 203 | 11001011 | SET6, (IX+00) | DD 221 | 11011101 | |
| RST 10 | D7 215 | 11010111 | | CE 206 | 11001110 | | DF 223 | 11011111 | | CB 203 | 11001011 | |
| RST 18 | DF 223 | 11011111 | SET1, A | CB 203 | 11001011 | SET3, B | CB 203 | 11001011 | | 00 0 | 00000000 | |
| RST 20 | E7 231 | 11001111 | | CF 207 | 11001111 | | D8 216 | 11011000 | | F6 246 | 11101110 | |
| RST 28 | EF 239 | 11101111 | SET1, B | CB 203 | 11001011 | SET3, C | CB 203 | 11001011 | SET6, (IY+00) | FD 253 | 11111101 | |
| RST 30 | F7 247 | 11101111 | | CB 200 | 11001000 | | D9 217 | 11011001 | | CB 203 | 11001011 | |
| RST 38 | FF 255 | 11111111 | SET1, C | CB 203 | 11001011 | SET3, D | CB 203 | 11001011 | | 00 0 | 00000000 | |
| SBC A, (HL) | 9E 158 | 10011110 | | C9 201 | 11001001 | | DA 218 | 11011010 | | F6 246 | 11101110 | |
| SBC A, (IX+00) | DD 221 | 11011101 | SET1, D | CB 203 | 11001011 | SET3, E | CB 203 | 11001011 | SET6, A | CB 203 | 11001011 | |
| | 9E 158 | 10011110 | | CA 202 | 11001010 | | DB 219 | 11011011 | | F7 247 | 11101111 | |
| | 00 0 | 00000000 | SET1, E | CB 203 | 11001011 | SET3, F | CB 203 | 11001011 | SET6, B | CB 203 | 11001011 | |
| SBC A, (IY+00) | FD 253 | 11111101 | | CB 203 | 11001011 | SET3, G | CB 203 | 11001011 | | F0 240 | 11110000 | |
| | 9E 158 | 10011110 | SET1, H | CB 203 | 11001011 | | DC 220 | 11011100 | | SET6, C | CB 203 | 11001011 |
| | 00 0 | 00000000 | | CC 204 | 11001100 | SET3, L | CB 203 | 11001011 | | F1 241 | 11110001 | |
| SBC A, A | 9F 159 | 10011111 | SET1, L | CB 203 | 11001011 | SET4, | DD 221 | 11011101 | SET6, D | CB 203 | 11001011 | |
| SBC A, B | 98 152 | 10011000 | | CD 205 | 11001101 | | CB 203 | 11001011 | | F2 242 | 11110010 | |
| SBC A, C | 99 153 | 10011001 | SET2, (HL) | CB 203 | 11001011 | | E6 230 | 11100110 | | | | |
| | | | | D6 214 | 11010110 | | | | | | | |

Index

| | | | | | |
|------------|--------|----------|-------------|--------|----------|
| SET7, A | CB 203 | 11001011 | SRL (IX+00) | DD 221 | 11011101 |
| | FF 255 | 11111111 | | CB 203 | 11001011 |
| SET7, B | CB 203 | 11001011 | | 00 0 | 00000000 |
| | FB 248 | 11111000 | | 3E 62 | 00111110 |
| SET7, C | CB 203 | 11001011 | SRL (IY+00) | FD 253 | 11111101 |
| | F9 249 | 11111001 | | CB 203 | 11001011 |
| SET7, D | CB 203 | 11001011 | | 00 0 | 00000000 |
| | FA 250 | 11111010 | | 3E 62 | 00111110 |
| SET7, E | CB 203 | 11001011 | SRL A | CB 203 | 11001011 |
| | FB 251 | 11111011 | | 3F 63 | 00111111 |
| SET7, H | CB 203 | 11001011 | SRL B | CB 203 | 11001011 |
| | FC 252 | 11111100 | | 3B 56 | 00111000 |
| SET7, L | CB 203 | 11001011 | SRL C | CB 203 | 11001011 |
| | FD 253 | 11111101 | | 39 57 | 00111001 |
| SLA(HL) | CB 203 | 11001011 | SRL D | CB 203 | 11001011 |
| | 26 38 | 00100110 | | 3A 58 | 00111010 |
| SLA(IX+00) | DD 221 | 11011101 | SRL E | CB 203 | 11001011 |
| | CB 203 | 11001011 | | 3B 59 | 00111011 |
| | 00 0 | 00000000 | SRL H | CB 203 | 11001011 |
| | 26 38 | 00100110 | | 3C 60 | 00111100 |
| SLA(IY+00) | FD 253 | 11111101 | SRL L | CB 203 | 11001011 |
| | CB 203 | 11001011 | | 3D 61 | 00111101 |
| | 00 0 | 00000000 | SUB(HL) | 96 150 | 10010110 |
| | 26 38 | 00100110 | SUB(IX+00) | DD 221 | 11011101 |
| SLA A | CB 203 | 11001011 | | 96 150 | 10010110 |
| | 27 39 | 00100111 | | 00 0 | 00000000 |
| SLA B | CB 203 | 11001011 | SUB(IY+00) | FD 253 | 11111101 |
| | 20 32 | 00100000 | | 96 150 | 10010110 |
| SLA C | CB 203 | 11001011 | | 00 0 | 00000000 |
| | 21 33 | 00100001 | SUB A | 97 151 | 10010111 |
| SLA D | CB 203 | 11001011 | SUB B | 90 144 | 10010000 |
| | 22 34 | 00100010 | SUB C | 91 145 | 10010001 |
| SLA E | CB 203 | 11001011 | SUB D | 92 146 | 10010010 |
| | 23 35 | 00100011 | SUB 00 | D6 214 | 11010110 |
| SLA H | CB 203 | 11001011 | | 00 0 | 00000000 |
| | 24 36 | 00100100 | SUB E | 93 147 | 10010011 |
| SLA L | CB 203 | 11001011 | SUB H | 94 148 | 10010100 |
| | 25 37 | 00100101 | SUB L | 95 149 | 10010101 |
| SRA(HL) | CB 203 | 11001011 | XOR(HL) | AE 174 | 10101110 |
| | 2E 46 | 00101110 | XOR(IX+00) | DD 221 | 11011101 |
| SRA(IX+00) | DD 221 | 11011101 | | AE 174 | 10101110 |
| | CB 203 | 11001011 | | 00 0 | 00000000 |
| | 00 0 | 00000000 | XOR(IY+00) | FD 253 | 11111101 |
| | 2E 46 | 00101110 | | AE 174 | 10101110 |
| SRA(IY+00) | FD 253 | 11111101 | | 00 0 | 00000000 |
| | CB 203 | 11001011 | XOR A | AF 175 | 10101111 |
| | 00 0 | 00000000 | XOR B | A8 168 | 10101000 |
| | 2E 46 | 00101110 | XOR C | A9 169 | 10101001 |
| SRA A | CB 203 | 11001011 | XOR D | AA 170 | 10101010 |
| | 2F 47 | 00101111 | XOR 00 | EE 238 | 11101110 |
| SRA B | CB 203 | 11001011 | | 00 0 | 00000000 |
| | 2B 40 | 00101000 | XOR E | AB 171 | 10101011 |
| SRA C | CB 203 | 11001011 | XOR H | AC 172 | 10101100 |
| | 29 41 | 00101001 | XOR L | AD 173 | 10101101 |
| SRA D | CB 203 | 11001011 | | | |
| | 2A 42 | 00101010 | | | |
| SRA E | CB 203 | 11001011 | | | |
| | 2B 43 | 00101011 | | | |
| SRA H | CB 203 | 11001011 | | | |
| | 2C 44 | 00101100 | | | |
| SRA L | CB 203 | 11001011 | | | |
| | 2D 45 | 00101101 | | | |
| SRL(HL) | CB 203 | 11001011 | | | |
| | 3E 62 | 00111110 | | | |

A to D converter, 316, 353–5
 ABS, 75
 absolute jumps *see* test and jump set
 accumulator, 387, 389, 399–404
 acoustic coupler, 341
 ACS, 75
 address, 264
 addressing: direct, 390
 immediate, 389
 indirect, 391
 PC-relative, 392
 addressing methods, 387–97
 aerial splitter, 7, 8
Alchemist, The, 253
 allophones, 313–14
 Alphacom printer, 329
 alternate registers, 367
 amplitude (of sound), 129
 analogue to digital converter *see* A to D converter
 AND, 367, 369, 400–2
 animation, 83, 301
 apostrophe, 24
 argument, 33
 arithmetic instructions, 20
 arithmetic set (MPU), 267
 arithmetic signs, 20, 30
 array, 44–9
 ASCII codes, 28, 35, 71, 264, 340, 366, 370, 415
 ASN, 75
 assembler, 376, 378, 425
 assembler, 437–45
 assembly language, 377, 388–27
 assignation, 27–8
 assignation, complex, 372
 assignation, simple, 372
 AT, 21
 Atari D-type plug, 276–7
 ATN, 75
 ATTR, 99–100
 attributes, 79
 automatic boot, 343
 azimuth adjustment, 299, 300
 base address, 396
 BASIC, 17, 67
 BASIC syntax, 17
 Basicare system, 356
 baud rate, 298, 341, 343
 BC registers, 391
 BEEP, 80, 129
 BEEP amplifiers, 299
 BIN, 95
 binary codes, 85–6, 261, 377
 binary-hex conversion, 379–80
 bit, 261
 block diagram, 265
 BORDER, 5, 8, 95–6
 brackets, 39
 BRIGHT, 92–3
 buffer, 306, 369
 buses, 265
 byte, 25, 262
 byte, loading a, 266
 byte, storing a, 266
 CALL, 420
 CAPS LOCK, 18
 CAPS SHIFT, 18
 cassette recorder, 10–11, 271, 297–8
 CAT, 303
 Centronics interface, 332
 channels, 304–6
 character ratings (in games), 248–50
 character set (in ROM), 373
 characters (in games), 247–51
 CHR\$, 35, 52, 81, 84, 264
 CIRCLE, 91
 circles, 101–7
 CLEAR, 404–5
 clock-pulse generator, 375

CLOSE, 307, 344
 CLS, 20, 29
 CODE, 35, 87, 425
 codes, 263–4, 279
 colon, 25
 colours, 95–100
 combat routine (in games), 250
 comma, 22
 command, immediate, 10
 commands (in games), 234–6
 composite video, 349
 conditional jump instruction, 395
 control devices, 316, 353–6
 COS, 102
 CP, 400, 403
 crashing through, 58
 credit cards, use of, 460
 cube (of a number), 21
 cursor, 17, 73

D & D *see* *Dungeons and Dragons*
 D-type plug (Atari) *see* Atari
 DATA, 40, 53, 55
 data, selection of, 58
 data, sorting, 58
 data bus, 387
 data bytes, 376
 data entry (from keyboard), 287
 data filing, 52
 data pins (MPU), 268
 data processing, 51
 data transmission, 338
 DE registers, 391
 debugging, 437–9
 DEC, 400
 DEF FN, 60–62
 defined function *see* DEF FN
 DELETE, 13
 demodulation, 349
 denary, 264, 377
 denary-hex conversion, 381–4
 digital tracers, 321, 323–5
 DIM, 44
 directory (disk), 309
 disassembler, 378, 419
 disk drive units, 310
 disk drives, 297, 308–11
 displacement byte, 392
 display file (in RAM), 373
 DJNZ, 422–4
 DOS *see* operating system

dot crawl, 349
 DRAW, 87–92
Dungeon Builder, 231–2
 dungeon master, 220
Dungeons and Dragons, 219
 duplex (for modems), 339

ECG monitor, 355
 ellipses, 111–13
 ENTER, 17, 52
 EPROMs, 355
 ERASE, 304
 event table (in games), 233, 239–41
 Expansion slot *see* User port
 experience points, 248–50

feed mechanisms, printer, 331
 filing on disk, 309
 firmware, 273
 flag register *see* status register
 FLASH, 92–3
 flowcharts, 413–15
 FN, 60–62
 FOR, 41
 FORMAT, 303, 344
 frequency (of sound), 129
 function *see* DEF FN, FN

game playing (using keyboard), 287
 garbage, 341, 363
 gates, 375, 387
 GOSUB, 51
 GOTO, 37
 graphics pads, 321, 327–8
 graphics plotters, 331

hardware, 375
 hex codes, 377–84
 hex conversion *see* binary, denary
 hexadecimal codes *see* hex codes
 hexagons, 107
 high resolution graphics, 81, 87–92, 98, 101–27
 HL registers, 391
Hobbit, The, 222
 hypotenuse, 102

Ice King, The, 224–6, 234–44
 IF, 37
 IN, 282–3, 354, 427
 INC, 400

increment, 56
 index registers, 396–7
 initialisation, 363
 INK, 79, 95–7
 INKEY\$, 42, 52, 346
 input, 270–72
 INPUT, 28, 45, 307, 344
 INPUT LINE, 32
 instruction bytes, 376
 INT, 35
 integer, 18, 366
 integer, negative, 367
 interface, 263, 272
 joystick, 279–82
 parallel, 332
 printer, 332
 serial, 332
 TV, 273
 video, 350–51

Interface 1 (Sinclair), 288, 301, 305
 Interface 2 (Sinclair), 280
 INVERSE, 92
 inverse character, 81
 italic symbols, 124–6
 IX register, 396
 IY register, 396

joystick, analogue, 275–7
 joystick, switch, 275, 276
 JR instructions *see* jump-relative
 instructions, 392
 jump set (MPU), 267–8

Kempston I/F, 279, 281, 282
 Key (on peripherals), 6, 258
 keyboard, 282–5

least significant digit *see* significance
 LEN, 33, 245
 LET, 27
 Light Rifle (Stack), 2, 79
 light pens, 321–2
 line number, 18
 line overhead, 371
 LLIST, 75
 LN, 76
 LOAD., 344, 425
 LOAD, 21, 55, 91, 303
 locations (in games), 225–6, 232, 237
 locking up, 258, 345
 logic set (MPU), 267, 269

loop, 38, 41, 42, 368, 421 4, 439
 loop, endless, 376
 loops, nested, 46
 low resolution graphics, 81–4, 97–8
 LPRINT, 75

mail order, 459–61
 maps (for games), 226–8
 memory kits, testing, 294
 memory upgrade kits, 291–6
 menu (in programs), 51, 63
 MERGE, 57, 67, 438
 message table (games), 232, 236–7
 microchip, 291
 microchips, fitting, 293
 Microdrive, 297, 301–8
 Micronet 800, 337, 342
 Microprocessor Unit *see* MPU
 Middle C, 130
 mnemonics, 388
 modem, hardwire, 341
 modems, 338–43
 modulation, 349
 monitor (video), 349–51
 monitors (programs), 439–40
 most significant digit *see* significance
 MOVE, 307, 346
 movement table (in games), 231, 238
 MPU, 266, 271, 375–411
 mugtraps, 41, 46
 multi-statement line, 25, 371
 music, 131 3
 music program, 133
 music synthesisers, 316–19

negative numbers, 384
 networking, 343–7
 NEXT, 41

objects (in games), 232, 238–9, 251
 octagons, 109
 OPEN, 306, 344

operand (assembly language), 389
 operating system, 310
 operator (assembly language), 389
 OR, 267, 269, 400, 402
 OUT, 314, 317, 354, 427
 outline (for games), 224
 output, 270–72
 OVER, 93

page address *see* base address
 paged memory, 295
PAPER, 79, 95–7
 paper, printer, 335
 parity bit, 339
PAUSE, 42, 54
 PC register, 387
 PCB, 288–9, 293
PEEK, 74, 264, 377
 peripherals, 257
 perspective drawing, 326
PI, 103
 pitch, 319
 pixels, 79
PLOT, 87–9
POINT, 99, 119–20
POKE, 74, 86, 193, 365, 377, 421
 polygons, 107
POP, 406, 432
 port, 270, 425–35
 position independent, 426
 potentiometer, 275, 323
 power supply, 5
 Prestel, 337, 342
PRINT, 17, 24, 306, 344
 printed circuitboard *see* PCB
 printers, daisywheel, 330, 334
 dot matrix, 330
 full-width, 330–31
 ink jet, 331
 thermal, 330
 Prism VTX5000 modem, 339, 342–3
 program, 10, 63–72
 program, branching, 37
 Program Counter *see* PC register
 program entry (with keyboard), 287
 programs, testing of, 66, 72, 243
 programs, machine code, 406–35
PUSH, 406, 432

Quill, The, 231–4
 quote marks, 18

 radians, 91, 103
RAM, 263, 363
 RAMpacks, plug-in, 295
 random filing (disk) *see* filing on disk
 random monsters (in games) 473
RANDOMIZE, 71, 405
READ, 40
 read signal (in MPU), 387

read-write head (disk), 309
 reading (MPU operation), 268
 register (sound unit), 317–19
 registers (in MPU), 387–406
 relative jumps *see* test and jump set
REM, 137, 404, 441
RESTORE, 43
RET, 420
RETURN, 51
 RGB video, 349
 ribbon cables (from keyboard), 288–9
 ribbons, printer, 335
RND, 45
ROM, 263
 ROM cartridges, 280, 282
ROTATE, 353–5, 399–401
 rotation of figures, 113–17, 121–3
 RS232 interface, 301, 332, 340
RUN, 17
 RUN-time actions, 372

SAVE, 11, 21, 53, 55, 87, 92
SAVE., 303, 344, 425
 scaling, 111, 114, 123–6
Scroll?, 47
 sectors (disk), 309
 semicolon, 22
 sensors, 354
 sequential filing (disk) *see* filing on disk
SGN, 75
SHIFT, 399–401
 sign bit, 384
 signed number, 386
 significance, 262
SIN, 91, 101
 sketching programs, 126–8, 279
 Small Claims procedure, 462
 software, 273, 375
 sound synthesisers *see* music synthesisers
 speech recognition, 315–16
 speech units, 313–14
SQR, 30
 square (of a number), 21
 square root *see* SQR
 stack, 51, 406
 Stack Light Rifle *see* Light Rifle
 start bit, 339
 static electricity, 292
 status register, 394
 status table (games), 233, 241–3
STEP, 41

stepper motors, 355
Stick, Le, 279
 stop bit, 339
STR\$, 34
 streams, 304–8
 string, 27, 83
 string packing, 245–7
 string slicing, 32–3
 strings, comparison of, 35–6
SUB, 418
 subroutines, 66–7, 363
 subscript, 44
 switching units, 354
SYMBOL SHIFT, 18

TAB, 21
 Telesoftware, 343
 Teletext, 349
 Television, 6, 8–10, 349–50
 test and jump set, 403
THEN, 38
 through bus, 282
 time controller, 355
TO, 33, 41
 tokens, 264, 340, 374
 tracks (disk), 309
 traps (in games), 252
 Trickstick, 278
 tuning (TV), 8–10
 two's complement, 385

 unsigned number, 386

user-defined graphic, 84–7
 User port, 6, 349–50
USR, 86, 405, 421

VAL\$, 78
VAL, 34, 52
Valhalla, 225
 variable, 27–8
 variable, initialising, 39
 variable, local, 62
 variable, non-integer, 367
 variable, number, 30–32, 365
 variable, string, 31–32, 367
 Variable List Table *see* VLT
VERIFY, 21, 54, 87
VID strap, 350
 viewdata, 337
VLT, 364, 365
 vocabulary (games), 232, 235–6

 warbling note (trill), 130
 waveform (of sound), 129, 319
 word processing, 286
 writing (MPU) operation, 268

 X-rays, 351
XOR, 267, 269, 400, 402

 Z80A microchip, 266
 ZX printer, 329, 331
ZX81, 74

| SOFTWARE GUIDES | |
|---|----------------------|
| WORKING WITH dBASE II | 0 246 12376 1 £7.95 |
| USING YOUR MICRO | |
| COMPUTING FOR THE HOBBYIST AND SMALL BUSINESS | 0 246 12023 1 £6.95 |
| DATABASES FOR FUN AND PROFIT | |
| 0 246 12032 0 £5.95 | |
| FIGURING OUT FACTS WITH A MICRO | 0 246 12221 8 £5.95 |
| INSIDE YOUR COMPUTER | |
| 0 246 12235 8 £4.95 | |
| SIMPLE INTERFACING PROJECTS | 0 246 12026 6 £6.95 |
| PROGRAMMING | |
| COMPLETE GRAPHICS PROGRAMMER | 0 246 12280 3 £6.95 |
| THE COMPLETE PROGRAMMER | 0 246 12015 0 £5.95 |
| PROGRAMMING WITH GRAPHICS | 0 246 12021 5 £5.95 |
| WORD PROCESSING | |
| CHOOSING A WORD PROCESSOR | 0 246 12347 8 £7.95 |
| WORD PROCESSING FOR BEGINNERS | 0 246 12353 2 £5.95 |
| FOR YOUNGER READERS | |
| BEGINNERS' MICRO GUIDES: ZX SPECTRUM | 0 246 12259 5 £2.95 |
| BEGINNERS' MICRO GUIDES: BBC MICRO | 0 246 12260 9 £2.95 |
| BEGINNERS' MICRO GUIDES: ACORN ELECTRON | 0 246 12381 8 £2.95 |
| MICROMATES | |
| SIMPLE ANIMATION | 0 246 12273 0 £1.95 |
| SIMPLE PICTURES | 0 246 12269 2 £1.95 |
| SIMPLE SHAPES | 0 246 12271 4 £1.95 |
| SIMPLE SOUNDS | 0 246 12270 6 £1.95 |
| SIMPLE SPELLING | 0 246 12272 2 £1.95 |
| SIMPLE SUMS | 0 246 12268 4 £1.95 |
| GRANADA GUIDES: COMPUTERS | 0 246 11895 4 £1.95 |
| MEMOTECH | |
| MEMOTECH COMPUTING | 0 246 12408 3 £5.95 |
| THE MEMOTECH GAMES BOOK | 0 246 12407 5 £5.95 |
| ORIC-1 | |
| THE ORIC-1 AND HOW TO GET THE MOST FROM IT | 0 246 12130 0 £5.95 |
| THE ORIC PROGRAMMER | 0 246 12157 2 £6.95 |
| THE ORIC BOOK OF GAMES | 0 246 12155 6 £5.95 |
| TI99/4A | |
| GET MORE FROM THE TI99/4A | 0 246 12281 1 £5.95 |
| VIC-20 | |
| GET MORE FROM THE VIC-20 | 0 246 12148 3 £5.95 |
| THE VIC-20 GAMES BOOK | 0 246 12187 4 £5.95 |
| ZX SPECTRUM | |
| AN EXPERT GUIDE TO THE SPECTRUM | 0 246 12278 1 £6.95 |
| INTRODUCING SPECTRUM MACHINE CODE | 0 246 12082 7 £7.95 |
| LEARNING IS FUN: 40 EDUCATIONAL GAMES FOR THE SPECTRUM | 0 246 12233 1 £5.95 |
| MAKE THE MOST OF YOUR ZX MICRODRIVE | 0 246 12406 7 £5.95 |
| THE SPECTRUM BOOK OF GAMES | 0 246 12047 9 £5.95 |
| SPECTRUM GRAPHICS AND SOUND | 0 246 12192 0 £6.95 |
| THE SPECTRUM PROGRAMMER | 0 246 12025 8 £5.95 |
| THE ZX SPECTRUM AND HOW TO GET THE MOST FROM IT | 0 246 12018 5 £5.95 |
| WHICH COMPUTER? | |
| CHOOSING A MICROCOMPUTER | 0 246 12029 0 £4.95 |
| LANGUAGES | |
| COMPUTER LANGUAGES AND THEIR USES | 0 246 12022 3 £5.95 |
| EXPLORING FORTH | 0 246 12188 2 £5.95 |
| INTRODUCING LOGO | 0 246 12323 0 £5.95 |
| INTRODUCING PASCAL | 0 246 12322 2 £5.95 |
| MACHINE CODE | |
| Z80 MACHINE CODE FOR HUMANS | 0 246 12031 2 £7.95 |
| 6502 MACHINE CODE FOR HUMANS | 0 246 12076 2 £7.95 |
| APPLE II | |
| APPLE II PROGRAMMER'S HANDBOOK | 0 246 12027 4 £10.95 |
| AQUARIUS | |
| THE AQUARIUS AND HOW TO GET THE MOST FROM IT | 0 246 12295 1 £5.95 |
| ATARI | |
| GET MORE FROM THE ATARI | 0 246 12149 1 £5.95 |
| THE ATARI BOOK OF GAMES | 0 246 12277 3 £5.95 |
| BBC MICRO | |
| ADVANCED MACHINE CODE TECHNIQUES FOR THE BBC MICRO | 0 246 12227 7 £6.95 |
| ADVANCED PROGRAMMING FOR THE BBC MICRO | 0 246 12158 0 £5.95 |
| THE BBC MICRO: AN EXPERT GUIDE | 0 246 12014 2 £6.95 |
| BBC MICRO GRAPHICS AND SOUND | 0 246 12156 4 £6.95 |
| DISCOVERING BBC MICRO MACHINE CODE | 0 246 12160 2 £6.95 |
| DISK SYSTEMS FOR THE BBC MICRO | 0 246 12325 7 £7.95 |
| HANDBOOK OF PROCEDURES AND FUNCTIONS FOR THE BBC MICRO | 0 246 12415 6 £6.95 |
| INTRODUCING THE BBC MICRO | 0 246 12146 7 £5.95 |
| LEARNING IS FUN: 40 EDUCATIONAL GAMES FOR THE BBC MICRO | 0 246 12317 6 £5.95 |
| TAKE OFF WITH THE ELECTRON AND BBC MICRO | 0 246 12356 7 £5.95 |
| 21 GAMES FOR THE BBC MICRO | 0 246 12103 3 £5.95 |
| PRACTICAL PROGRAMS FOR THE BBC MICRO | 0 246 12405 9 £6.95 |
| THE COLOUR GENIE | |
| MASTERING THE COLOUR GENIE | 0 246 12190 4 £5.95 |
| COMMODORE 64 | |
| BUSINESS SYSTEMS ON THE COMMODORE 64 | 0 246 12422 9 £6.95 |
| ADVENTURE GAMES FOR THE COMMODORE 64 | 0 246 12412 1 £6.95 |
| COMMODORE 64 COMPUTING | 0 246 12030 4 £5.95 |
| COMMODORE 64 DISK SYSTEMS AND PRINTERS | 0 246 12409 1 £6.95 |
| THE COMMODORE 64 GAMES BOOK | 0 246 12258 7 £5.95 |
| COMMODORE 64 GRAPHICS AND SOUND | 0 246 12342 7 £6.95 |
| IBM | |
| THE IBM PERSONAL COMPUTER | 0 246 12151 3 £6.95 |
| LYNX | |
| LYNX COMPUTING | 0 246 12131 9 £6.95 |