The COMPLETE

PART 6

SPECTRUM

All you want to know about the world's best-selling computer

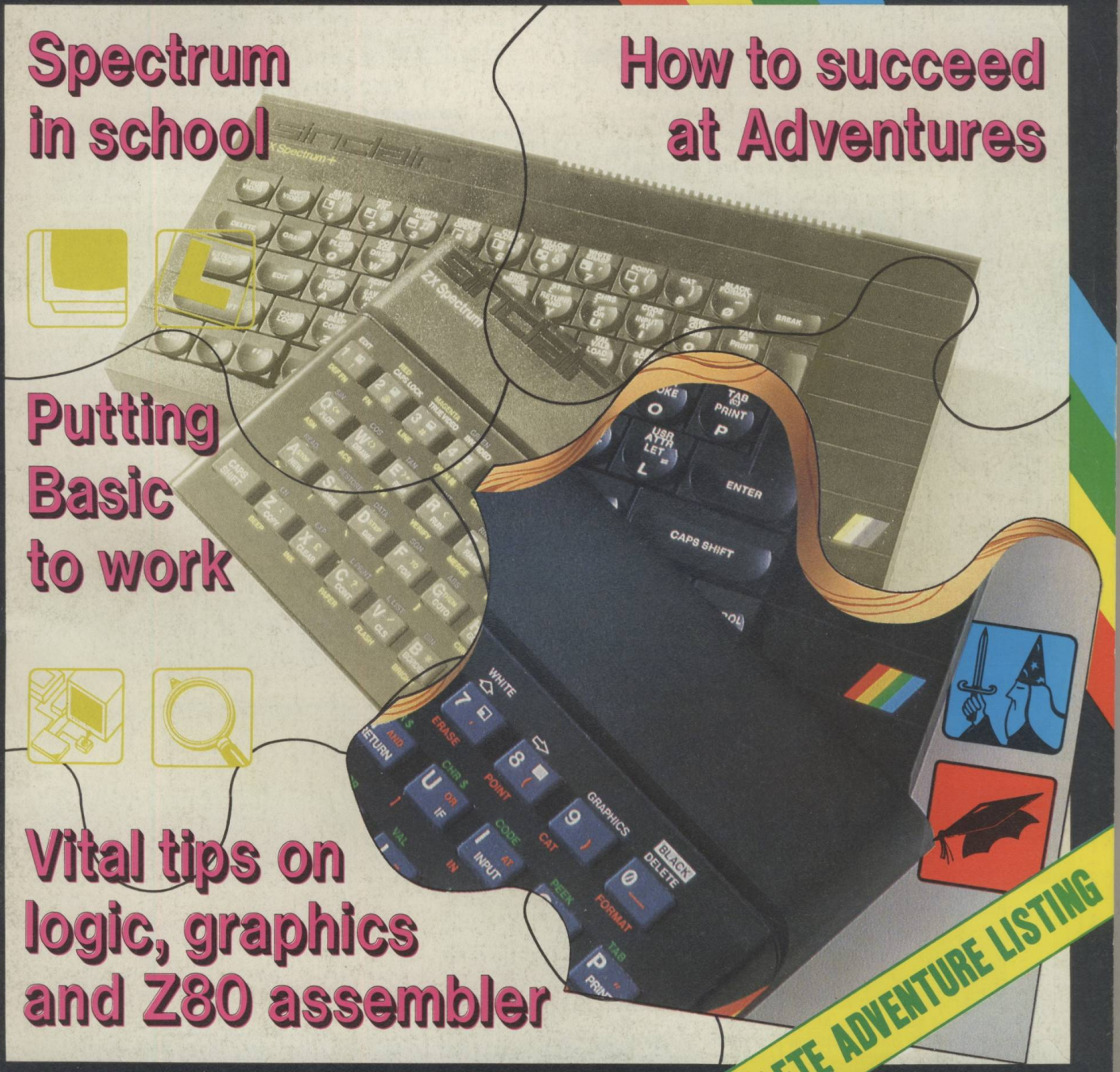
A Database Publication

Spectrum in school

How to succeed at Adventures

Putting Basic to work

Vital tips on logic, graphics and Z80 assembler



COMPLETE ADVENTURE LISTING

In six monthly parts

£1.50

Issue by issue The Complete Spectrum has been building up to be the ultimate reference work on Spectrum computing . . .

Part 1



Your first 30 minutes ● Saving and loading advice Avoiding errors when typing in listings ● The Z80 chip investigated ● Computer jargon clarified ● Simple guides to Basic, machine code and graphics.

Part 2



Discovering discs ● Microdrives ● Upgrading to 48k Composite video ● Graphics commands ● Books Animation techniques ● Plus Basic, machine code and graphics.

Part 3



The 128k Spectrum ● Printers and interfaces Databases ● Speech synthesis ● Keyboard add-ons Graphics commands ● Animation techniques ● With Basic, machine code and graphics tutorials.

Part 4



Inside the 128 ● The mouse ● Joysticks ● Graphics packages ● Spreadsheets ● Machine code graphics ● With basic Basic, machine code and animation secrets.

Part 5



Prestel and Micronet ● Modems and electronic mail Hacking ● How Basic works ● Lightpens and graphics tablets ● Machine code graphics ● Structured programming ● The 128k Spectrum's Basic ● Plotters Plus Basic and machine code tutorials.

Please use the order form on the inside back cover

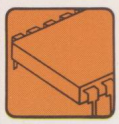
In Part Six...



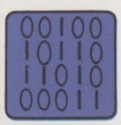
182 Beginners
More for beginners: Two dimensional arrays and conditional statements explored.



186 Spectrum in School
How the Spectrum adds impetus to learning, and the techniques for effective software.



192 How Basic Works
Our investigation of Basic continues with a look at how variables are stored in memory.



194 Binary Breakthrough
Practical uses of AND, OR and EOR round off our series on using binary numbers.



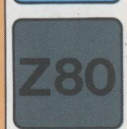
198 Into Adventures
What makes micro-adventures so fascinating? An introduction to this exciting world.



202 Creating Adventures
Want to write your own adventure? Here's a few hints plus a testing little game.



206 Top Adventures
In a detailed survey of top adventure games we discover there's something for everyone.



209 Machine Code
We show you how to make full use of the input and output routines based in the ROM.



212 Program Style
Put the finishing touches to Smiley Hunt, a structured program explained in detail.



214 Machine Code Graphics
We put theory into practice and provide the routines to move a sprite around the screen.



216 Index
A comprehensive summary of the contents of all six sections of this partwork.

The COMPLETE SPECTRUM
All you want to know about the world's best-selling computer

is published by
Database Publications Ltd,
Europa House, 68 Chester Road,
Hazel Grove, Stockport SK7 5NY.
Tel: 061-456 8383

© Database Publications Ltd. No material may be reproduced in whole or in part without written permission. While every care is taken, the publishers cannot be held responsible for any errors in articles, listings or advertisements.

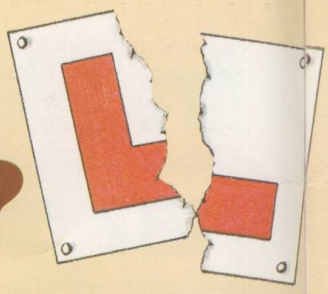
News trade distribution by:
Europress Sales & Distribution Ltd,
Unit 1, Burgess Road, Ivyhouse Lane,
Hastings, East Sussex TN35 4NR.
Tel: 0424 430422.

The Complete Spectrum has been written by: Mike Bibby, Pete Bibby, Henry Budgett, Mike Cook, Mike Cowley, Iolo Davidson, Kevin Edwards, Peter Freebrey, Terry Greer, John Hughes, Alan McLachlan, Roland Waddilove.

Design: Heather Sheldrick. **Photography:** Paul Francis.

Illustration: Gordon Brookes, Bill Worthington, Pamela Dunkerley, Tim Leckey.

The end of the beginning?



The last part of the series that aims to explain the elements of Basic for absolute beginners

REMEMBER our six runners from last time? Until now we've used two arrays to store the details about age and time, one for each. When we talked about time we looked at *t( )*, when it was details of the runners' ages we wanted we looked at *a( )*. We switched between arrays as needed. However it is possible to hold both lists of data in one array, only now the subscript holds two numbers.

The first number is the "switch" which tells us which set of information we want. The second number is just a pointer to the particular bit of data we're after. Program I shows one of these two dimensional arrays, as they are known, in action.

```
 10 REM Program I
 20 DIM (2,6)
 30 FOR p=1 TO 2
 40 FOR q=1 TO 6
 50 READ r(p,q)
 60 NEXT q
 70 NEXT p
 80 PRINT "Pick time/age 1/2"
 90 INPUT choice
100 FOR n=1 TO 6
110 PRINT r(choice,n)
120 NEXT n
130 DATA 20,21,25,27,30,33
140 DATA 35,35,55,20,27,24
```

Program I

The DIM of line 20 works just as before except now it dimensions a two dimensional array. There are 12 elements of this array (2 times 6) but they're not numbered from 1 to 12 as was the case in our previous, one dimensional arrays. In fact there are two one dimensional arrays parallel to each other, each holding six items of data. The second dimension refers to the fact that to get at a particular element you have to first choose a one dimensional array and then specify which element of it you want to deal with.

Incidentally, just because it is a two dimensional array doesn't mean you're stuck to a choice between two arrays. You can have as many as you want, always provided that your Spectrum's memory doesn't run out. So you could have:

DIM w(3,9)

dimensioning a two dimension array of 27 elements or:

DIM d(5,50)

producing an array of 250 elements (five parallel arrays each holding 50 elements). For the moment, though, we'll stick to our simple array with 12 elements.

If you look at the data lines of Program I you should be able to see where the figures that are going to fill the array come from. The first six are the runners' times, the second their ages.

The nested FOR . . . NEXT loops should hold no fears for you. The outer loop – control variable *p* – circles twice. For each turn of the outer loop the inner loop – control variable *q* – cycles six times. Think what effect this will have on the READ line:

50 READ r(p,q)

At the beginning *p* is 1 so, as the inner loop cycles, values are read into the elements *r(1,1)* to *r(1,6)*. You'll see that this will read the times into those elements with 1 as the first figure in the brackets. Once this is done the second loop puts the data for the ages into the elements from *r(2,1)* to *r(2,6)*. So if you want to talk about times, the first number inside the brackets is 1. If it's ages you're after then it should be 2. Figure I shows the structure and contents of *r( )* after the program has been run.

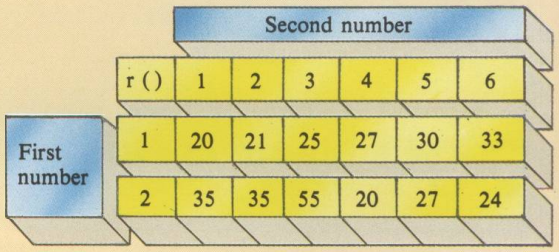


Figure II: The array r( ) and its contents

Once the data is actually in the array it's easy to manipulate, as the final loop shows. Here it gives you the choice of which data you get, the times or the ages. It's all there held in one array, *r( )*, available at the drop of a FOR . . . NEXT loop – provided that you have the appropriate numbers in the subscript.

Explore the two dimensional array that's been created. Try lines like:

PRINT r(1,3)
PRINT r(2,2)

until you can see how it's built up. And then, if you're feeling ambitious, try using two dimensional arrays in your own programs. You'll find that they are powerful ways of summarising – in a very flexible way – data given in table or row/column form.

And now string arrays

Having dealt with numeric arrays it should come as no surprise to learn that we can have string arrays. Like numeric arrays, the names of string arrays have to be

just one letter, with the dollar sign, $, tagged on so the Spectrum knows it's dealing with strings, not numbers.

The main difference between the two is that while numeric arrays hold numbers in their elements, string arrays hold characters. Let's set up a string array with:

```
DIM g$(5)
```

We now have a one dimensional string array consisting of elements *g$(1)*, *g$(2)*, on to *g$(5)*. Each of these can hold a character so let's fill them with:

```
LET g$(1)="a"
LET g$(2)="!"
LET g$(3)="+"
LET g$(4)="M"
LET g$(5)="7"
```

You can now retrieve each character using the relevant number in the subscript. So:

```
PRINT g$(3)
```

gives you a plus sign while:

```
PRINT g$(5)
```

returns 7.

This is just as it was with numeric arrays. However string arrays have another trick, just for themselves. Try:

```
PRINT g$
```

and you'll see:

```
a!+M7
```

appear on screen. So, with string arrays, you can not only address each element separately but you can use the root name to get all the elements at once. And this is the way that string arrays are mostly used. Set up another string array with five elements using:

```
DIM d$(5)
```

At the beginning, each element is filled with a space – corresponding to the elements of numeric arrays being initialised to zero. Now give it a string to hold with:

```
LET d$="KEVIN"
```

Notice that we're referring to the whole array by its root name. This doesn't worry the Spectrum which takes the string and slots it into the five elements of the array, character by character. Figure IIIa shows the characters in their array positions.

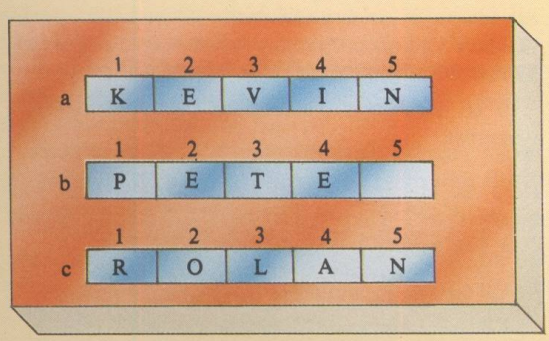


Figure III: The string array d$

Now store PETE in the array with:

```
LET d$="PETE"
```

This has only four characters, yet the array has five elements. What happens? The answer is that the "left over" elements are filled with spaces. In this case, as shown in Figure IIIb, there's only one space padding out the array.

Of course it could happen that the string being assigned to the array is too long, such as:

```
LET d$="ROLAND"
```

which attempts to put a six letter string into a five element array. Something's got to give and ROLAND gets his end lopped off as Figure IIIc shows. The moral is, always make sure your arrays are long enough to take all the characters in the strings you want them to hold.

It all depends

Have you noticed how relentless the Spectrum is? It starts at the beginning of a program and works through it line by line, obeying each and every line. A FOR . . . NEXT loop might send it round the houses but each line in a program is obeyed at least once.

However sometimes you don't want this to happen. It can happen that you only want a line to be obeyed if a certain condition is true. A bank manager might want his computer to tell him if a customer is overdrawn. But he only wants the line:

```
PRINT c$;" is in the red"
```

obeyed when *c$* really is in the red. If not then that line is not to be obeyed.

Spectrum Basic allows for this type of thing by having the IF . . . THEN statement. It works along the lines of:

```
IF condition is true THEN do something
```

evaluating the condition if, and only if, the condition is found to be true, obeying the rest of the line after the THEN. If the condition isn't true then all the code after the THEN is ignored.

Figure IV shows the comparisons that can be used line in Program II to:

Operator	Rule
=	Equal to
<>	Not equal to
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to

Figure VI: Operators and their rules

```
10 REM PROGRAM II
20 FOR n=1 TO 10
30 READ number
40 IF number=5 THEN PRINT number; "is 5!"
50 NEXT n
60 DATA 0,1,2,3,4
70 DATA 5,6,7,8,9
```

Program II

The comparison is made in line 40. Here the contents of a variable, *number* are compared with the number 5. It's easy to see that if *number* holds 5 then the condition is true (5=5) and the rest of the line is obeyed. Notice that as the FOR . . . NEXT loop is obeyed, 10 numbers are read into *number*. However it's only when the condition is true (when *number*=5) that the message is printed.

Following close on the comparative operator = (as these things are known) is <>. You'll find it on the W key. This is the opposite of = and means, unsurprisingly, "is not equal to". Try changing line 40 of Program II to:

```
40 IF number<> 5 THEN PRINT number;"
is not five"
```

and you'll see what it does. Now the condition is true whenever *number* is not equal to 5. When this is the case the message is displayed. When *number* does hold 5 then the condition can't be true (how can you have 5 not equal to 5?) and the rest of the line is ignored. So you get a message for every value of *number* but 5.

The next operator we'll meet is < which means "less than". You'll find it sharing the R key. Again, alter the line in Program II to:

```
40 IF number < 5 THEN PRINT number;"
is less than 5"
```

and see what happens. Now the rest of the line after the THEN is only obeyed when *number* holds a value less than 5. The result is that the program gives messages for values of 0 to 4 but none after that. Don't get the "less than" operator, <, confused with the "less than or equal to" operator <=. Found on the Q key, this is subtly different in its effect. It also allows the case where *number* is equal to as well as less than 5. If you try:

```
40 IF number <= 5 THEN PRINT number;"
is either less than 5 or equal to 5"
```

in Program II you'll see that now you get the numbers 0 to 5 displayed.

Since we've had a "less than" operator it seems logical that there should be a "greater than" operator. There is, it's >, found on the T key. Using:

```
40 IF number > 5 THEN PRINT number;"
is greater than 5"
```

will give you 6, 7, 8, 9 as your reward, all the values of *number* that are greater than 5.

Not surprisingly there's also a "greater than or equal to" operator. It's >=, to be found lurking on the E key. A quick alteration to Program II in the form of:

```
40 IF number >= THEN PRINT number;"
is either greater than 5 or equal to 5"
```

will show you what it does. Now the output is 5, 6, 7, 8, 9. The "equal to" part of the condition has accepted the case where *number* is 5.

Try altering the comparisons made in line 40, using different values. Also, you can change the numbers in the data lines and see what happens. Be sure to note the difference between < and <=, > and >=. It may seem small but can be a potent force for program faults.

Notice that the operators are in opposites. If something is <= to something else, it can't be > than it. That is, if *number* is less than or equal to 5, say 4, then it obviously isn't greater than 5. Similarly if it's greater than or equal to 5, say 6, then it can't be less than 5. The conditions are mutually exclusive. Don't worry if this seems a bit academic. When you come to use IF . . . THEN to get your programs to take decisions, you'll understand.

In the above programs we've just compared the value in a variable with a number. We could also compare two variables or an expression such as:

```
IF oneVariable > twoVariable THEN . . .
IF 2*oneVariable < 3*twoVariable THEN . . .
```

Also the examples chosen have used the IF . . . THEN conditional statement with just a PRINT after the THEN. There can in fact be all kinds of keywords after the THEN such as LET or CLS. You can even send the program hurtling all over the place as we'll see shortly. Which ever way they are used, IF . . . THEN statements are extremely important. They allow our programs to make choices and so become much more flexible and useful.

Two at a time

Often we don't want to test for just one condition, we want to test for two. For example, looking at Program II again, we may want a number that lies between 4 and 7. Here there are two conditions to be considered. We want *number* to be greater than 3 and at the same time *number* has to be less than 8.

If you use:

```
40 IF number>3 AND number<8 THEN
PRINT number
```

you'll see how this is done. All we've done is to join our two conditions with a logical operator, AND. Now both conditions have to be true before the code after the THEN is obeyed. The result is that 4, 5, 6 and 7 are returned.

Try the following line:

```
40 IF number>=3 AND number<=8 THEN
PRINT number
```

Can you see how both conditions combine to give 3, 4, 5, 6, 7 and 8? While we often want two conditions both to be true before we go on to the bit after the THEN, sometimes we want the line to be obeyed if one or other or both of two conditions are true. We may want to go out if it's sunny or dry or both. Here only one of the conditions has to be true for us to be out of the door. The only way we stay in is if both conditions are false – if it's wet and dark.

There's a logical operator to deal with this, the aptly named OR. You can see it in action with:

```
40 IF number<3 OR number>7 THEN PRINT
number
```

Here the line is obeyed if *number* is less than 3 or greater than 7. The result is that 0, 1, 2, 8 and 9 appear. Can you explain why:

```
40 IF number<=3 OR number>=7 THEN
PRINT number
```

gives 0, 1, 2, 3, 7, 8 and 9? As with single comparisons,

try changing Program II to try different combinations of conditions acting on different numbers.

String comparisons

Just as we can compare numbers and numeric variables, so we can compare strings. This is possible because, as you know, computers work with numbers. Micros store strings as numbers and when we ask our Spectrum to compare, say, A with B it compares the numbers it holds for the strings.

Happily we don't have to be concerned with the numbers, we can just use the strings with the comparison operators. All we have to know is that a space has the lowest number, the numbers come next, then capital letters then lower case letters. And we can think of the < operator meaning "comes before" and > as meaning "comes after". So comparing strings you should see that:

```
a>1
a>A
a<z
a>Z
1<9
ab<ac
aa>aA
```

are all true. You'll notice that some of the above are more than one character long. To compare two strings the Spectrum looks at the first character. If these are the same then it goes on to the next character, if any.

Program III not only shows you string comparisons in action, it also lets you explore the way the Spectrum orders strings. Play around with it until you've grasped how it does it. And then, if you're really keen to test what you've learnt, try writing a program that will take five strings and display them in alphabetical order.

```
10 REM Program III
20 PRINT "Enter two strings"
25 INPUT a$,b$
30 IF a$<b$ THEN PRINT a$;" comes before "
;b$
40 IF a$>b$ THEN PRINT a$;"comes after ";b$
50 IF a$=b$ THEN PRINT " The strings are
the same"
```

Program III

Subroutines at work

When we combine subroutines with IF . . .THENs we can get some very powerful structures. Program IV does just this.

Although in this case the subroutines are fairly trivial the point to grasp is that each is completely different and you choose which is to be performed. So one program could do three or more different things at your whim. In the real world, where the subroutines might contain some quite powerful code, this makes them both powerful and flexible.

Another property of subroutines is that they allow programs to be planned and coded by dividing them into simple steps. Take a look at Program V.

This impressive bit of code doesn't do anything! It's just a dummy but it reflects a program that might be written – one that gets information, processes it and gives the result.

However once you've got a dummy like this, the problem has been broken down into four areas, each of which can be worked on separately. If you wanted you could deal with the *getInfo* subroutine, leaving the rest of the program as it is.

You might even break this down into several subroutines and solve their coding one at a time. It really makes things easier doing it this way.

Try writing a program like this yourself. Once you've seen how the old principle of "divide and conquer" works in programming it's unlikely that you'll try any other method.

```
10 REM Program IV
20 LET first=200
30 LET second=300
40 LET third=400
50 PRINT "Polite/Complimentary/Rude
1/2/3"
60 INPUT choice
70 IF choice=1 THEN GO SUB first
80 IF choice=2 THEN GO SUB second
90 IF choice=3 THEN GO SUB third
100 STOP
200 REM Polite
210 PRINT "It's nice to meet you"
220 RETURN
300 REM Complimentary
310 PRINT "It's wonderful to meet you"
320 RETURN
400 REM Rude
410 PRINT "Meeting you is the pits"
420 RETURN
```

Program IV

```
10 REM Program V
20 LET start=100
30 LET getInfo=200
40 LET doSomething=300
50 LET giveResults=400
60 GO SUB start
70 GO SUB getInfo
80 GO SUB doSomething
90 GO SUB giveResults
99 STOP
100 REM initialise
110 PRINT "start"
120 RETURN
200 REM get information to work on
210 PRINT "getInfo"
220 RETURN
300 REM process the information
310 PRINT "doSomething"
320 RETURN
400 REM display the results
410 PRINT "giveResults"
420 RETURN
```

Program V

The end and the beginning

And this is where our course on beginner's Basic ends. Yet it's only the start. Although we've covered the basics of Spectrum Basic there's a lot more to learn. And even if you don't bother going deeper into the language, you know enough to write some very satisfying and practical programs. So, you've got the knowledge, you've got the Spectrum. The rest of it is really up to you.

Spectrum in schools



This low cost micro can provide a high degree of learning impetus in the classroom

IT is a bit of a mystery as to why the Spectrum hasn't achieved a higher penetration into English schools. Statistics published by the Microelectronics Education Programme showed that the BBC Micro dominates the English educational field. This isn't the case in Scotland, where the Sinclair machines, both ZX81 and Spectrum, had a much bigger impact. There, the use in education reflects quite closely the overall market share of the Sinclair machines.

The reasons for this revolve around three areas – the keyboard, microdrives and lack of educational software. But the Sinclair Spectrum has a valid role in all areas of education.

Let's start with the keyboard of the Spectrum series. The original "dead flesh" keyboard did not inspire confidence when compared with the real keyboards of the BBC Micro or Electron. However, Spectrum Plus and 128 machines are a little better, and I've even heard some teachers say that the old style keyboard was easier for some children to use, as the keys were separated from each other on the keyboard by a distance sufficient to ensure that two or more keys weren't pressed at once.

Microdrives have been generally accepted now as a mistake. Educationalists were quite rightly concerned about the long term reliability of such an innovative piece of technology, and the high price of blank cartridges and their relative vulnerability were big problems.

Finally, software. Education authorities buying computers are effectively buying engines on which to run educational software. In the classroom a large software base is vital, and the Spectrum has never had this in the education field, despite the huge quantity of programming talent that exists for the machine. Perhaps one of the reasons for lack of software development on the Spectrum is that the machine, when running in Basic, is painfully slow when compared with the BBC Micro. This has led to much educational software for the Spectrum involving more machine code routines than a corresponding piece of software on the BBC Micro, thus increasing development time and costs.

In addition, Spectrum Basic doesn't possess some of the features of BBC Basic, such as decent run-time error trapping – again features which can be added from machine code. Lastly, the graphics on the Spectrum suffer from the way in which the colour attributes are related to the character square rather than the pixel.

But these technical problems can all be overcome. There has always been a "chicken and egg" situation with regard to computers and their software. As long as there aren't many machines in education publishers will obviously think twice before investing time and money in developing software for a small market. However, if the software doesn't exist, the user base of the machine is unlikely to get any bigger. As we'll soon see, some software houses and users have been brave enough to try and break this cycle.

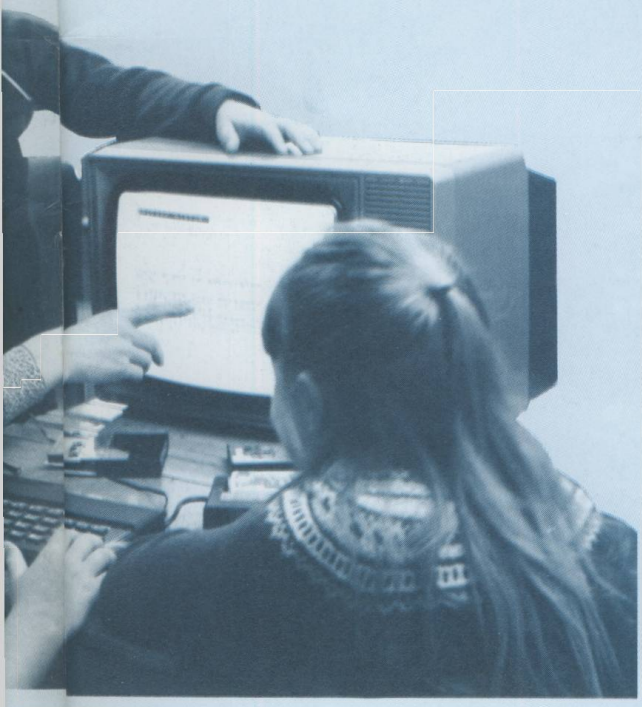
The plus points

It mustn't be forgotten that the Spectrum offers some good advantages to the educationalist, starting with cost, and this is continued with the later versions of the machine. For the price of a single BBC Micro it is possible to get two or three Spectrum systems. This offers a better pupil to computer ratio, giving the children obvious advantages – in any given period more children can be using a computer.

The larger amount of memory in a Spectrum also allows larger programs to be used without the need for disc systems. Finally, the use of a computer which has received a high home penetration might facilitate the eventual development of software and resources for education that are as usable in the home as they are in the classroom. Of course, care needs to be taken here to develop educationally valid materials that can be used without the guidance of a teacher, but this is an interesting field for future work.

Down to work

So, having set the scene, what can schools do with Spectrums? Well, the answer is virtually anything that can be done on the BBC Micro. Software does exist for the Spectrum, but you have to search around for it. A useful address for teachers interested in the educational use of the Spectrum is Resource, Exeter Road, off



Coventry Grove, Doncaster DN2 4PY. This organisation, funded by some of the South Yorkshire Local Education Authorities, has produced Spectrum in Schools newsletters and good quality Spectrum software. In addition, various other software houses have versioned their programs to run on the Spectrum and a vast industry consisting of companies producing a range of add-on peripherals for the Spectrum has sprung up.

Primary level

Let's start with primary education. Logo has rightly become an important educational tool in this area. This "turtle graphics" language allows children to use the computer as a tool with which to explore various concepts, especially those of a mathematical nature. Sinclair produced a full version of Logo, including the list processing features of the language as well as the more usual turtle graphics. It's not a cheap package, but the educational advantages offered by Logo are extremely valuable.

Also, word processors like Primary Pen, a primary oriented word processing package, allow the Spectrum to be used as a creative writing tool. Other word processors, such as Tasword, can also be pressed into service. Of course, word processors need printers for output, and here a variety of companies have produced software and hardware to allow the Spectrum to drive Epson or similar printers.

A third type of software that has proved valuable at this level has been the database. Many teachers working with the Spectrum started off with Vu-File, and were able to introduce the database into project work, using the Spectrum to store all sorts of information. An early educational database for the Spectrum was Factfile, produced by MEP and published by Cambridge Microsoftware.

Resource has produced a series of well-received dedicated databases such as Birdwatch, which allows a database of information about birds to be accessed and updated by children with great ease. The package also includes a graphs pack, allowing the graphical display of information held in the database. Another is Library, which is designed to hold information about books, allowing children to search the database for books which may be of use in their current work.

Other types of primary educational software are also available for the Spectrum, covering a wide range of different fields. Software houses such as Blackboard Software, Sinclair-Macmillan and Ginn and Company have all produced some software in the past. The teacher wanting to use the Spectrum in education will, though, have a harder search for worthwhile material than his or her colleague using BBC Micros.

Databases down the telephone

The Spectrum educationalist will find no difficulty in accessing databases such as Prestel, Ceefax or Oracle. A variety of modems are available allowing access to the Prestel database which could easily provide a vast, if a little costly, information resource. In addition, Teletext decoders are available allowing access to what David Dodds, former headmaster of Thurcroft School in Rotherham, called Freefax – the pages of Ceefax and Oracle. He argued that the educational resource provided by these systems provides a useful source of national information that can be downloaded, printed out and used in project work.

Secondary level

As we move up in age the software availability decreases. Secondary level Spectrum software is nowhere near as widespread as the primary level material, so it's not surprising that relatively little work has been done with Spectrums in the general curriculum. However there is one area in which the Spectrum is strong at this level – control technology.

This is simply the design and construction of electronic control systems. These don't have to include computers, but many such systems do. Why should the Spectrum be so good at this? After all, it hasn't got the built-in user ports or analogue to digital converter of the BBC Micro. Well, it's cheaper, and control technology is one field where hands on experience is more valuable than a large and expensive computer. Indeed, much of the early work in this field was done using the Micro-Professor single board computer – a Z80-based system with a 20-key keyboard and an LED display.

A variety of companies, such as Griffin and George, produce input/output ports which allow the Spectrum to control a wide range of additional devices such as small motors, lights and other electronic circuits. For example, Griffin markets a device called I-Pack which provides TTL input and output ports for connection to other electronic circuits, switch inputs and relay outputs to allow simple switching of circuits and eight analogue inputs for voltage monitoring. Software for control applications is simple, so Sinclair Basic is perfectly adequate for the job.

However at least two languages exist for the Spectrum which have been designed to facilitate the easy development of sophisticated control applications on the Spectrum. The first of these is Control Basic, a fully integrated Basic extension. The second is Control Logo, a version of Logo with extra commands, produced by the Advisory Unit for Computer Based



Education of Hertfordshire LEA. Information on both these products can be obtained from Resource.

Related to the above is the use of the Spectrum as a data logger in the school laboratory. This is simplicity itself, requiring just an analogue to digital converter and a few sensors to convert various physical parameters such as light or temperature, into an electrical signal. Using a light sensor and the I-Pack device mentioned above, for example, it is possible to monitor the voltage and current characteristics of a small electric bulb as well as the light level, in the milliseconds after the light is turned on. Try that with a voltmeter and ammeter!

The intention of using the micro in this way is not to replace traditional methods of experimental school science, but is to open up a whole variety of new fields for investigation. Anything that can produce a voltage can be monitored, graphs can be drawn – even Fourier analysis of input signals can be done with suitable software. Frequency can be directly measured, short time delays measured, events counted and so on. The Spectrum, with a suitable interface and software, can replace a wide variety of expensive laboratory hardware.

Special educational needs

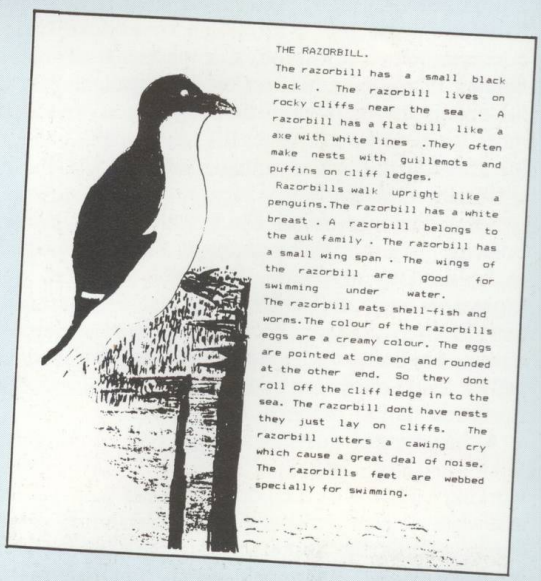
Special input devices aimed at children who have difficulties with standard keyboards, such as the Concept keyboard or Possum system, can be interfaced to the Spectrum, providing a cheap alternative to other micros.

Here the much maligned Spectrum keyboard wouldn't matter as it's not being used. Admittedly, though, the software situation in this area isn't very good for the Spectrum.

Conclusion

So there we have it. The Spectrum is capable of a vast range of uses in the classroom, from controlling robots or floor turtles to allowing pages of Ceefax information to be printed out. Spectrums have tended to sneak in to schools, often supplementing already existing BBC machines and providing useful hands on experience for children who might not otherwise have got it with, say, only one BBC Micro available.

Despite the demise of the old Spectrum, we should look at the Spectrum Plus and the Spectrum 128 as useful tools for the future, especially in the current educational economic climate. Now, all we need to do is twist the arms of a few software houses.



A week's field study in Yorkshire was recorded by a class of 10 year olds using the Tasword 2 word processing package. One feature of this program is that a window can be placed in the text to allow illustrations to be added.



3D GANGSTERS "HIT" YOUR SCREEN

SPECTRUM 48K
£7·95
SPECTRUM 48K

Interactive `bubble´ speech adds to the drama of this true life detective movie

Don't miss this picture - it speaks for itself!

AMSTRAD
£8·95
AMSTRAD

Imagine Software (1984) Limited • 6 Central Street • Manchester M2 5NS • Tel: 061 834 3939 • Telex: 669977

Imagine Software is available from: WHSMITH, John Menzies, WOOLWORTH, LASKYS, Rumbelows, Greens, Spectrum Shops and all good dealers.

QUICK TO LEAR

THAT'S...

FINALIST BRITISH MICRO COMPUTING AWARDS 1985

MINI OFFICE

SPREADSHEET



	A MONEY	B JANUARY	C FEBRUARY	D MARCH
2:	MORTGAGE	85.72	85.72	85.72
3:	FOOD	46.24	41.47	26.45
4:	FUEL	46.25	47.28	26.28
5:	LEISURE	20.00	20.00	20.00
6:	OTHER	99.85	17.12	56.23
8:	TOT SPENT	298.06	211.59	214.68
11:	EARNINGS	321.21	321.21	321.21
12:	B. FWD.	27.25	0.00	27.41
14:	TO SPEND	348.46	321.21	348.62
15:	SPENT	298.06	211.59	214.68
17:	REMAINING	0.00	109.66	113.95
19:	SAVE	0.00	82.25	85.46
20:	C.FWD.	0.00	27.41	28.49

JUST LOOK WHAT THIS PACKAGE CAN DO!

WORD PROCESSOR – Ideal for writing letters or reports! *Features:* Constant time display ● Constant word count (even shows words per minute) ● Normal or double-height text on screen or printout.

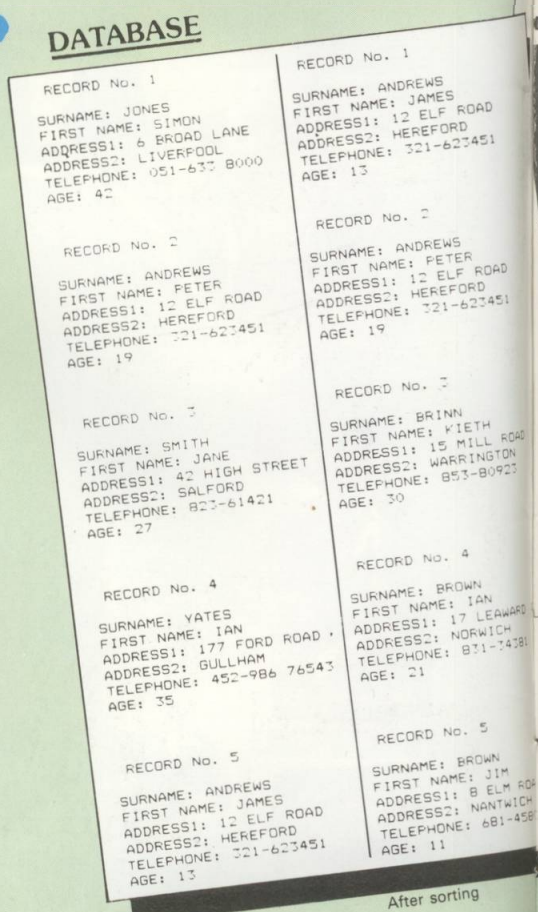
SPREADSHEET – Use your micro to manage your money! *Features:* Number display in rows and columns ● Continuous updating ● Update instantly reflected throughout spreadsheet ● Save results for future amendments.

GRAPHICS – Turn those numbers into an exciting visual display! *Features:* 3D bar chart ● Pie chart ● Graph.

DATABASE – Use it like an office filing cabinet! *Features:* Retrieve files at a keystroke ● Sort ● Replace ● Save ● Print ● Search.

The screen dumps shown are produced on a BBC Micro with an Epson printer. Results from other computers and printers may vary slightly depending on equipment used. Where narrow paper/printer is used similar results can be achieved by cut and paste.

DATABASE



```
RECORD No. 1
SURNAME: JONES
FIRST NAME: SIMON
ADDRESS1: 6 BROAD LANE
ADDRESS2: LIVERPOOL
TELEPHONE: 051-633 8000
AGE: 42

RECORD No. 2
SURNAME: ANDREWS
FIRST NAME: PETER
ADDRESS1: 12 ELF ROAD
ADDRESS2: HEREFORD
TELEPHONE: 321-623451
AGE: 19

RECORD No. 3
SURNAME: SMITH
FIRST NAME: JANE
ADDRESS1: 42 HIGH STREET
ADDRESS2: SALFORD
TELEPHONE: 823-61421
AGE: 27

RECORD No. 4
SURNAME: YATES
FIRST NAME: IAN
ADDRESS1: 177 FORD ROAD
ADDRESS2: GULLHAM
TELEPHONE: 452-986 76543
AGE: 35

RECORD No. 5
SURNAME: ANDREWS
FIRST NAME: JAMES
ADDRESS1: 12 ELF ROAD
ADDRESS2: HEREFORD
TELEPHONE: 321-623451
AGE: 13
```

Before sorting

```
RECORD No. 1
SURNAME: ANDREWS
FIRST NAME: JAMES
ADDRESS1: 12 ELF ROAD
ADDRESS2: HEREFORD
TELEPHONE: 321-623451
AGE: 13

RECORD No. 2
SURNAME: ANDREWS
FIRST NAME: PETER
ADDRESS1: 12 ELF ROAD
ADDRESS2: HEREFORD
TELEPHONE: 321-623451
AGE: 19

RECORD No. 3
SURNAME: BRINN
FIRST NAME: KIETH
ADDRESS1: 15 MILL ROAD
ADDRESS2: WARRINGTON
TELEPHONE: 853-80927
AGE: 30

RECORD No. 4
SURNAME: BROWN
FIRST NAME: IAN
ADDRESS1: 17 LEAWARD
ADDRESS2: NORWICH
TELEPHONE: 871-7481
AGE: 21

RECORD No. 5
SURNAME: BROWN
FIRST NAME: JIM
ADDRESS1: 8 ELM RO
ADDRESS2: NANTWICH
TELEPHONE: 681-458
AGE: 11
```

After sorting

...and it's all at price of just £

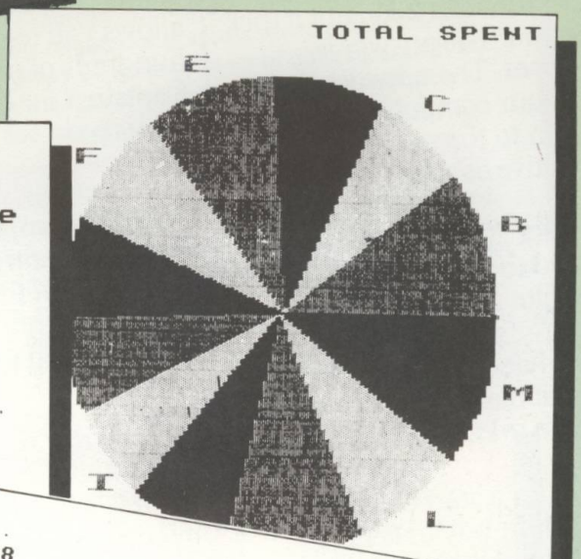
RN, EASY TO USE

Spectrum 128

Spectrum 48k

and Spectrum+

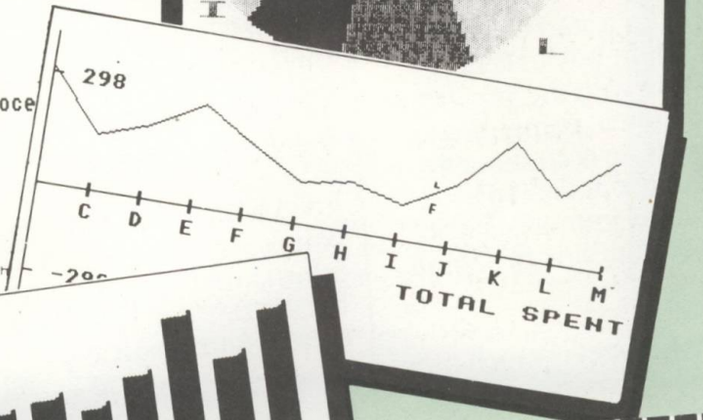
GRAPHICS

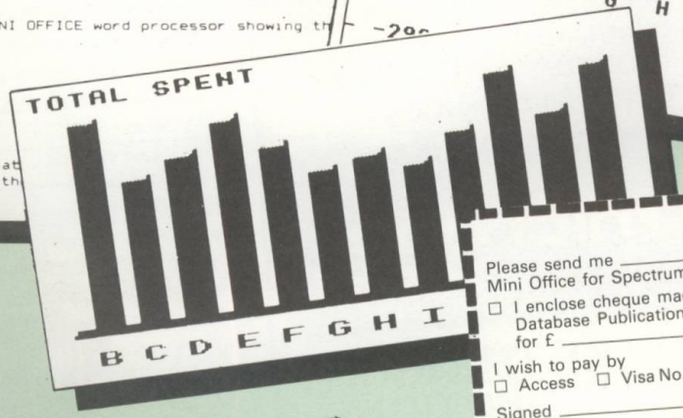


WORD PROCESSOR

Page 1

This is a demonstration of the MINI OFFICE word processor showing the various printout options available.





at the unbelievable

st £5.95 CASSETTE

DATABASE SOFTWARE

Please send me _____ copy/copies of Mini Office for Spectrum 48k/Spectrum+

☐ I enclose cheque made payable to Database Publications Ltd. for £ _____

I wish to pay by ☐ Access ☐ Visa No. _____ Expiry date _____

Signed _____

Name _____

Address _____

Post to: Mini Office Offer, Database Publications, 68 Chester Road, Hazel Grove, Stockport SK7 5NY.

Only £5.95

How does 20p a day turn your micro into a MAINFRAME?

Impossible you say! Just 20p a day for all that power?

No it's not a joke, nor a trick question. The answer is Micronet. The network that links your computer by modem to the most exciting interactive database in the U.K.

All for 20p a day subscription and the price of a local, cheap rate telephone call.*

So what does Micronet offer that has excited 20,000 people to join.

Well, for a start Micronet is an excellent way to keep up-to-date with the very latest information in computing. Our database is continually updated by our team of professional journalists who search out the stories behind the news.

But Micronet is much more than a news service – our revolutionary mainframe Chatlines give you the power to hold real-time conversations with up to 20,000 other users, instantly.

Our free national electronic mail system allows you to send and receive confidential mail to friends and businesses 24 hours a day.

You can even take part in the latest multi-user strategy games. Starnet for example, allows you to compete against hundreds of other "Star Captains" for domination of the galaxy.

Or win big weekly cash prizes by competing in the 'Round Britain Race' which makes use of the latest viewdata technology to challenge you to find secret locations.

Every day new *free* and discounted software is made available for downloading direct to your micro.

Teleshopping is the ultimate way to seek out high street bargains... or holiday and hotel bookings... computer dating... rail and airline information... Jobsearch... homestudy and schooling... ideas, information and entertainment facilities too numerous to list. As if all this wasn't enough you can also access Prestel's™ enormous database which is included in your subscription.

Micronet: the stimulating, challenging and informative way to add a whole new dimension to your micro. All you need is a modem. So cut the coupon today for your information pack.

For just 20p** a day can you afford to be left out in the power game?

micronet 800

MAKE THE RIGHT CONNECTIONS

*For 98% of telephone users.

Prestel is a trademark of British Telecommunications plc on Prestel.

Micronet 800, Durrant House, 8 Herbal Hill, London EC1R 5EJ. Telephone: 01-278 3143.

To: Micronet 800, Durrant House, 8 Herbal Hill, London EC1R 5EJ. Telephone: 01-278 3143.

Please rush me the full facts about Micronet and tell me how I can turn my micro into a mainframe for 20p per day. (£16.50 per quarter**).

NAME

ADDRESS

TELEPHONE AGE

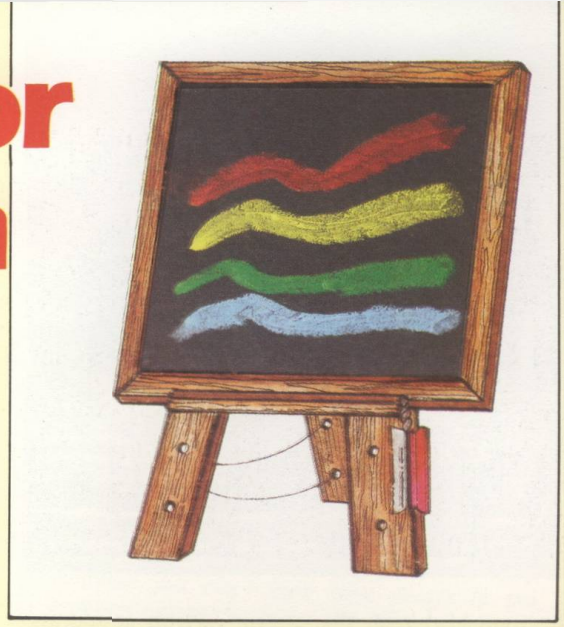
MAKE AND MODEL OF MICRO



CS

Criteria for education

Our experts share the benefit of their experience in that daunting field of educational programming



THE introduction of the micro into the classroom has led to a new breed of programmers. These are the ones who can create programs that not only provide an educationally valuable experience for the user but are also able to take anything an eager five or fifteen year old can throw at them and still carry on without crashing.

The generation of messages such as "Nonsense in Basic" or "No such variable" will, quite rightly, cause most teachers and many children to recoil in horror.

So let's examine the additional techniques a programmer needs to learn in order to produce robust educational software. However it's important to remember that all good education software is produced by a team including a teacher, an adviser and a programmer who has an intimate knowledge of the machine concerned.

There aren't many people who combine the ability to come up with a good educational idea and carry it through into a working program, so we'll look at the methods a programmer can use to increase the toughness of a program. The techniques will be explained with the aid of examples from Sinclair Basic, although the general points apply to programs written on any computer.

The screen – appearances count

Although what goes on the screen is specified by the teacher, there are a few points the programmer should be aware of.

1. The amount of information to be displayed on the screen at any one time is inversely proportional to the reading age and attention span of the child in question.

Reams of text for a five year old could be disastrous, as could one line per screen in an 'A' level program.

2. Use of colour and different sized text should not only take into account the age of the target child, but must also deal with technical problems, such as the relative readability of different coloured text on poor VDUs.

Many schools run their Spectrums on old TV sets, not RGB monitors.

3. Never break words at the end of lines. If the specification includes such screens, though, check with the teacher. There may be a valid educational reason, but it's doubtful.

4. A mixture of upper and lower case text on the screen is much more readable than upper case only.

Remember that the screen display is the main user interface with an educational program. A visually attractive screen will make the program more pleasurable to use. Use graphics where possible to add interest. Small things make a lot of difference, such as changing the border colour to that of the background paper, and so on.

Talking to the program

This is where most problems in poorly programmed Computer Assisted Learning (CAL) software turn up. We've probably all written programs that, for example, bombed out if we typed in 0 in reply to a prompt. (Come on you at the back, admit it!)

This is all very well for us, we know not to do it in future – but it's not at all satisfactory for a program that's to be used by a non-programming individual. Once we've got all the syntax and program logic errors out of a program, the only remaining cause for error is when user interaction brings in something unexpected. The motto of the educational programmer should be like that of a good Boy Scout – "Be prepared". In addition, remember Murphy's Law, clause 1, paragraph 1 – "If anything can go wrong, it will!"

A good program will not therefore, allow a bad input to the program – one that might cause a crash. This positive vetting is called input validation and is one of the most important items we'll discuss. So here we go . . .

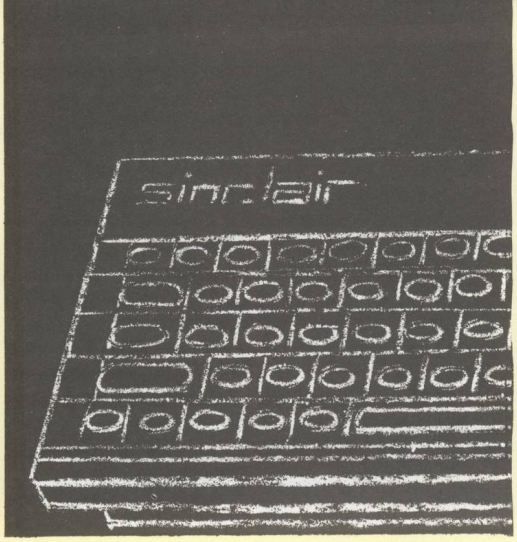
First of all, a couple of general points about the

Printers

Hard copy of a program's output is very desirable and a program should support at least one of the popular printers other than Sinclair's.

Printers such as the Epson require an interface but there are only a couple of really popular ones in circulation and these come with interface software to allow their use.

It will only improve your software if you write it in such a way that the user can easily patch in such a machine code program. This requires instructions, or course, but we will be providing comprehensive documentation . . . won't we?



Spectrum keyboard. The system variable at address 23609 decides the length of the click generated when a key is pressed. It's often a good idea to lengthen this by poking in a high value. This will make the user more aware of when a key is pressed. This pip is not generated when INKEY$ is used and in these cases we use the BEEP command, which also helps cut down accidental repeats of the key. While the BEEP beeps, the Spectrum ignores all else. A good statement to use is BEEP 0.1,1.

If the program is to be used by younger children it's often a good idea to disable the keyboard repeat altogether. This can be effectively done by POKEing a value of 0 into system variable REPDEL (23561). (23561). This causes a delay of about five seconds before a key starts repeating. Similarly, the value of REPPER (address 23562) can be adjusted to give a suitable delay between subsequent key repeats. A final useful trick is that if two or more keys are pressed at once, the INKEY$ function gets awfully flustered and returns nothing. It helps knowing that address 23560 holds the character code of the last key pressed. If two keys were pressed at the same time then the number in this address is the code of the first key of the two to be detected.

As for actually entering information into the machine, INPUT is never used in a good program. This statement allows the user to type in any amount of garbage and we only get to examine what is typed in after Enter has been pressed. The best approach is to use INKEY$ to get a character from the keyboard, then look at the character to see if it's acceptable.

As a simple start, let's look at the problem of getting a Yes/No reponse in reply to a question. The program must respond to Y or N, irrespective of the status of Shift, and so the simplest way of doing this task would be a subroutine such as the one in Program I.

```
1000 REM test for Y/N response
1010 LET temp=PEEK 23658
1020 POKE 23617,0: POKE 23658,8
1030 LET a$=INKEY$:IF a$="" THEN GOTO
     1020
1035 BEEP 0.1,1: IF a$<>"Y" AND a$<>"N"
     THEN GOTO 1020
1040 POKE 23658,temp
1050 RETURN
```

Program I: Getting a Yes/No response

Bit 3 of address 23658 specifies the case in which the keyboard operates. By POKEing it in line 1020 we're forcing the Spectrum to read/return an upper case letter whatever the status of the Caps Lock. Address 23617 specifies which of the Spectrum modes is selected. By POKEing it to 0 we specify the L mode.

The routine will return Y or N in *a$* depending on the keys pressed and will restore the contents of address 23658 to its original value. In addition, once entered, the only keys that will cause the routine to be left are Y, N or Break.

Similar routines can be used to check for other key presses. If you require the entry of, for example, a string of characters, we can write a subroutine that replaces INPUT, allowing us to check characters as they are typed in and reject them if they are not wanted. So we could enter a string of numbers by rejecting any character with an Ascii code of less than 48 and greater than 57. We could even check the actual number of characters entered, ensuring the string entered doesn't

Sound effects

While you may like the idea of rattling off two Brandenburg concertos or a quick "Ride of the Valkyries" while the program runs, don't forget that in a classroom only a small group will be using the computer.

Everyone else has to get on with other work and the noise from the computer can be a distraction. So it should be possible to turn off sound if necessary.

A simple way is to set a variable, say sflag *to 0 if sound isn't required and 1 if it is.*

Each BEEP statement used can thus be written as:

BEEP sflag*n,m

where n *and* m *are the BEEP parameters. Sound effects can reinforce the learning process, such as a fanfare for a correct answer. But don't go mad on these.*

Microdrive/disc drive

It should be possible for the user to transfer software from tape to microdrive easily. So no putting machine code in line 1 REM statements or other such nasty tricks.

File handling programs should be written with microdrive or tape systems in mind, as there doesn't appear to be a standard disc system in use for the Spectrum.

While on the subject of filing, suitable messages should be displayed when tape or microdrive operations are in progress. Users hate to be confronted by a system which is doing something they don't understand!

Messages such as "Please wait ... saving data" accompanied by some visual indication of how far the save (or load) operation has got do much to help the confidence of a teacher or child.

Help

It should be possible for a single key press to generalise some sort of help page full of useful information about the program, including any commands the program understands, and so on.

After this page the user can be returned to where he was in the program.

overwrite other parts of the screen display. Try doing that with the Basic INPUT statement!

Program II is a simple numeric input routine. It accepts positive integers only – a full number input routine would take care of decimal points and minus signs, ensuring that only one of each was allowed in a number. *y* and *x* specify the position of the input on the screen, *max* the largest number to be entered, *min* the smallest, and *chars* the number of characters to be accepted. After the routine has been used, the number entered will be in *num* and *n$*.

```
1000 REM number input routine
1010 REM s$ has more spaces than there
1015 REM will be digits in the number
1020 LET s$="      "
1030 PRINT AT y,x;s$(1 TO chars): LET n$=""
1040 LET c$=INKEY$: IF c$="" THEN GOTO
     1040
1050 LET c=CODE c$:BEEP 0.1,1
1060 IF (c<48 OR c>57) AND c<>13 AND c<>12
     THEN GOTO 1040
1070 IF c=13 AND n$<>"" THEN GOTO 1120
1080 IF c<>12 AND n$<>"" THEN LET n$=n$(1
     TO LEN n$-1): PRINT AT y,x;n$+" ":
     PRINT CHR$ 8
1090 IF LEN n$>chars THEN LET n$=n$(1 TO
     chars)
1105 PRINT AT y,x;n$
1110 GOTO 1040
1120 IF n$=CHR$ 13 THEN GOTO 1030
1125 LET num=VAL n$: IF num<min OR
     num>max THEN BEEP 0.5,2: GOTO 1030
1130 RETURN
```

Program II: A simple numeric input routine

As you can see, most of this routine is taken up with ensuring that a crash doesn't occur if the user tries to press Enter without entering a number, or if he tries to delete a non-existent character, and checking that the number entered is within the preset limits.

You could introduce further complexity – use your imagination. For a start, what about a box around the place on the screen at which data is to be entered, and a small cursor? But, of course, there is a trade off. The more error trapping we introduce the larger the program will be. However this is a small price to pay on the Spectrum with its relatively large memory.

Textual inputs can be processed in a similar fashion. For example, when asked for a filename you can check that it's the appropriate length and that it will be a legal one, as it's being entered. This prevents the Spectrum error messages being generated later in the program.

Additional advantages of this way of getting inputs are that you can print the keys to the screen in double size, different colours, and so on thus making the display more pleasant to use.

Error handling

Of course, there will always be a few errors that get through, such as the user pressing Break, a microdrive cartridge being absent when the program attempts to write to it, and so on. Most machines have something called ON ERROR GOTO which allows such events to be handled in a controlled fashion. The Spectrum doesn't, although it is possible to write machine code programs that perform such a function.

This is, however, beyond the scope of this article although it is something you should be aware of. If you validate your inputs and provide prompt messages, such as "Insert microdrive cartridge", at the appropriate time, you can at least prevent those errors due to forgetfulness or carelessness.

However if someone is determined to crash the program, the only prevention would be to implement machine code error trap routines. A program that crashes in this way, no matter how educationally sound it is, will get a quick reputation – of the wrong sort! It's rather ironic that the Spectrum, with a price that tempts the total novice, doesn't have a "nice" way of handling everyday errors.

Timing

No one could call the Spectrum a fast machine. However you can't have long delays in the program except for things like screen dumps to printers or tape/microdrive operations.

Any period of time longer than about 20 seconds with nothing apparently happening should be looked at with the idea of speeding up that section of the program – with machine code if necessary.

Check all the loops for statements that are redundant. Put all subroutines at the start of the program. Wherever possible avoid the use of trigonometric or logarithmic maths operations and see if there's simply a faster way of doing the job.

Sometimes, you won't be able to do anything about it, and it's often necessary to go back to the person who designed the program. Who knows, perhaps they may be able to redesign that section to cut down the delay?

End of lesson

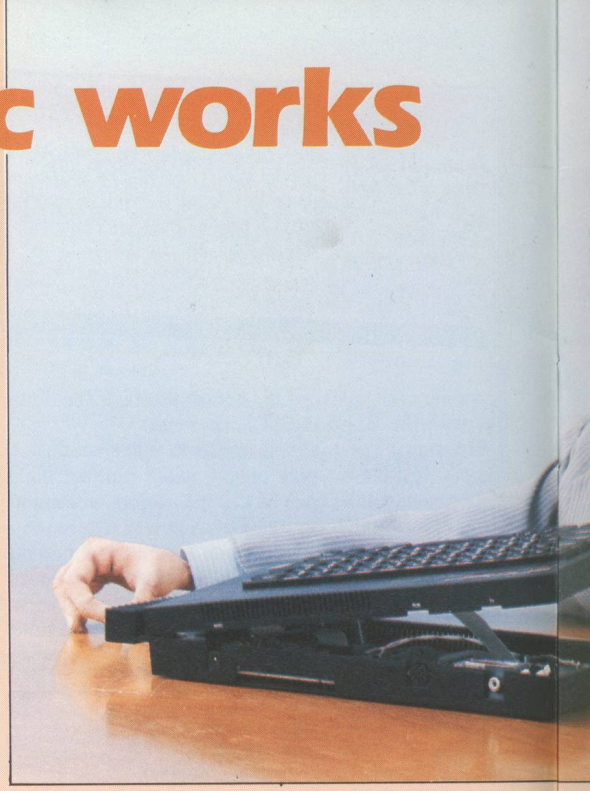
Finally there is the actual act of programming. The neater your program is, the easier it will be to debug. Do have a specification before you start, so you can plan the program. Structured programs of a sort are possible on the Spectrum, by using GOSUB, FN – and some care! GOTO statements should be used carefully. Those sending control of the program from, say, line 1 to line 2000 are generally caused by lack of thought at some point in the planning.

As for testing, the simplest thing is to do the silliest things you can think of, because you can guarantee that someone else will try it! Test each section of your program as you complete it and be prepared for bugs to turn up in the trial of the software with real kids.

Then, when it's all debugged, no one wants anything else added and you are totally happy with the program, drop us a line and tell us how on earth you did it!

How Basic works

Second and final part of the series on how your Spectrum interprets its own language



IN the first part of this series we saw how the lines of Basic that you input at the keyboard of your Spectrum are stored in RAM, beginning at a location specified by a system variable held in locations 23635 and 23636.

We also saw that the Basic keywords are represented by one-byte codes called tokens, which represent the various single-key commands.

Making a statement

Statements which involve numbers – such as assignments – are a little more complicated than those we looked at last time, especially as the Spectrum has various ways of handling numbers depending on whether they are integers or real.

To keep things simple, we shall be looking mainly at integers. Even so, you should pick up enough ideas to give you an insight into the general operating procedures of the machine, and to help you write more economical programs.

Briefly recapping, we found that when we entered the one-line program:

10 LET p=1

and then investigated the way in which it was stored, the computer responded by showing the sequences of numbers shown in Figure I.

You will notice that the number 1 is stored in two different ways – once for LISTing purposes (49 is the character code for 1) and once for the computer's own uses.

This is done by inserting the code 14 in the byte immediately after the code representing the number, and this acts as a signal to the Basic interpreter that the next five bytes are to be taken together, and that they contain a number.

All numbers on the Spectrum require five bytes for their representation, but the way in which this is done varies. If the number is an integer between +65535 and –65S35, the first byte holds a zero. The second holds zero for a positive number or 255 for a negative number – that is, it acts as a sign byte. The third and fourth bytes contain the actual value of the number (lo byte/hi byte), and the final byte holds zero again.

Real numbers are a bit more complicated. Briefly, the first byte contains the exponent of the number, but with 128 added to it, and the other four bytes are the mantissa. Zero is represented by all five bytes containing 0.

But don't worry if you don't understand any of that – just be grateful you are programming on the Spectrum and don't need to make the distinction between real numbers and integers, as you do on so many other computers!

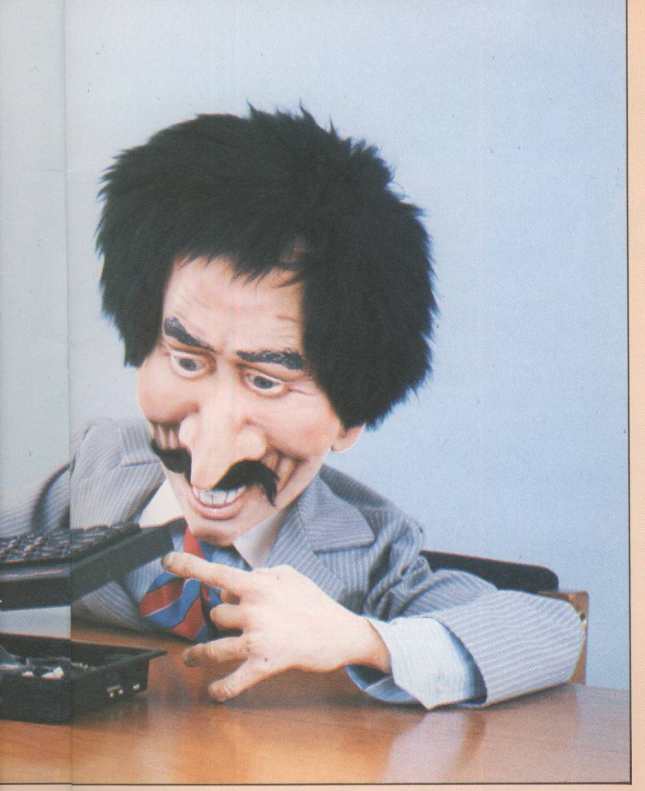
Something which is easily forgotten when working out how much memory a program will require is that the RAM has to hold both the program and a certain amount of working space. When your program is run, a number of extra locations in RAM are allocated to hold the variables. Information about this is contained in another systems variable, at locations 23627 and 23628.

To see how this works, type in NEW to get rid of your program, then enter the following:

```
10 LET a=1
20 PRINT a
30 STOP
40 FOR b=0 TO 5:PRINT PEEK (PEEK
   23627+256*PEEK 23628+b);" ";:NEXT b
50 LET a=a+1
60 GOTO 20
```

Line number	Line length in bytes	LET	P	=	1	Code for *number*	Internal representation of 1					Enter (end of line)
0 10	11 0	241	112	61	49	14	0	0	1	0	0	13

Figure I: How a line of Basic is stored



When the program is RUN, it will print 1 at the top of the screen and then wait for you to CONTinue. When you do, you will see the following on screen beneath the 1:

97 0 0 1 0 0 2

Disregarding the last number – which is the new value of *a* being printed as the program returns to line 20 – the other values represent the way in which the variable *a* is held in the variables store, the location of which is found by the PEEKs in line 40.

Figure II may make the sequence a little clearer. Incidentally, all variable names are treated as being lower case on the Spectrum, so if the variable had been *A* instead of *a*, the representation at this point would have been exactly the same.

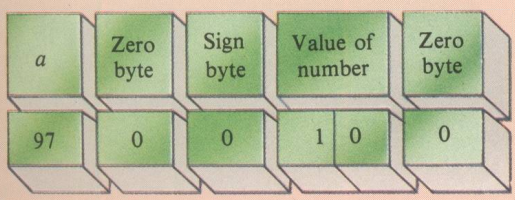


Figure II: How a is stored in memory

CONTinue once more and the display will change to read:

97 0 0 2 0 0 3

with the final number giving the latest value of *a* and the others showing how that value is updated in the variables storage area every time the program executes line 50.

Only single-letter variable names are stored in this simple way. For longer names the complete name is stored, but with 64 added to the value of the code of the first character and 128 to the value of the last.

This is a great advance on those machines which regard only the first two letters of a variable name as significant, but it does mean that if you are not careful, you can spend a lot of room storing unnecessarily lengthy variable names.

FOR . . . NEXT loops

The Spectrum recognises many different kinds of variable, and stores each kind in a slightly different way. Those controlling FOR . . . NEXT loops are perhaps the most interesting as all the information required to organise the loop is held in just 18 bytes.

Here the first byte contains the code for the variable name, with 128 added to it. The fact that only one byte is allocated explains why loop variables can have only a single-letter name.

The next 15 bytes are shared between the start value of the loop, the end value and the size of the STEP – five bytes each in the order indicated. The last two bytes contain the line number, stored in the usual hi byte/lo byte way.

Pointers

There are several useful pointers to be picked up from our look at the Spectrum's handling of Basic. The first is that because it is always possible to tell just where any particular line is stored in RAM, lines can be used for purposes other than the ones for which they were designed.

REM statements in particular make excellent stowing places for numbers which will form part of machine code instructions – they can be extracted with a PEEK, and then POKEd into the appropriate place while the Basic program is running.

The only thing to be careful about here is that you must know exactly at what address the statement is stored, as an error here will almost certainly result in the program crashing.

You can also use what you have learned here to cut down on the workspace requirements of your programs. Remember that unnecessarily inventing new variable names rapidly swallows up storage space, as each name is stored twice – once in the program line and once in the workspace.

Secondly, keeping variable names short is doubly efficient, as it saves room in both program and workspace. Of course, both of these tips conflict with the equally good advice to make your programs readable and easy to debug – the choice is yours.

Finally, what happens when you type NEW? In some computers, getting your program back is tolerably straightforward as it is not actually erased from memory.

The BBC Micro has the command OLD to restore the program without any fuss. The Spectrum doesn't do this – once you have scratched your program from RAM, then unless you have previously SAVEd it, it's gone for good. If you try that now, and then re-enter an immediate command to check up on it, you'll see what I mean.

What's that? Not gone completely – there's a little of it still there? I'm afraid not. What you are seeing is the representation in memory of the immediate mode Basic command which you've just entered!

Memories are made of bits

Conclusion of the series that helps take the mystery out of the way your Spectrum operates

SO far we have learned a lot about the binary system – the numbers our micro works in. We have seen that its memory is divided up into bytes – a set of eight two-state, binary units called bits. Each bit can have the value 1 or 0. If a bit has the value 1 we say it is set. If a bit has the value 0 we say it is clear.

As we're dealing with eight bits at a time we can use various combinations of the bits in a byte to code any whole number (integer) in the range 0 to 255. To do this we associate a code number with each bit. Figure I shows the scheme.

Our eight bits are labelled from b7 to b0 and the numbers associated with each number are shown above each bit (the more mathematical among you will see that they're in ascending powers of two).

To discover the value coded in a byte we simply add the numbers associated with every bit that is set (1), ignoring all clear bits (0). So

%10101000

codes the number:

128 + 32 + 8 = 168

We also learned to do tricks with, or to put it more properly, manipulate, binary numbers. We could create the complement of a number – a sort of binary opposite – by changing every clear bit to set ("setting" the bit) and changing every set bit to clear ("clearing" the bit).

So the complement of the number:

%10101000

gives us:

%01010111

We can add and subtract binary numbers, as well as multiply and divide. We learned other ways of combining them too, with the logical operators AND, OR, EOR. EOR, which stands for Exclusive OR, is also called XOR.

When combining two binary numbers under the influence of these operators we compare each bit in one number with the corresponding bit of the other.

Then, according to a rule which depends on the operator we're using, we decide whether that particular bit (the result bit) in the "answer" byte is set or clear. Table I shows the rules for the operators.

AND	Sets the result bit only if both bits compared are set, otherwise the result bit is clear.
OR	Sets the result bit if either or both the bits compared are set. Only if both bits compared are clear is the result bit clear.
EOR	Sets the result bit if the bits being compared differ in value. If the EOR bits compared are identical, the result bit is cleared.

Table I: Rules for logical operators

Memory, machines and maps

As we've said, a micro's memory is divided into byte-sized compartments, called memory locations. Each location has a number associated with it so we know which one we're talking about. These numbers are known as memory addresses.

Much of what a microprocessor does involves moving information – in the form of binary numbers – from one location to another. If you cast your mind back to earlier articles, I said that each bit was like a switch – its two values 1 and 0 could be used to signify that the switch was on or off respectively.

Imagine that we could wire up one of our bits to a machine's on/off switch. Then by setting that bit we could switch the machine on, and by clearing it we could switch it off.

This sort of thing is possible, though we'd need to use some clever electronics. In fact, since we deal with eight bits at a time, we could arrange things so that a single byte controlled the on/off status of eight separate machines – each machine m7,m6 . . . m0 corresponding to an individual bit of that byte, b7,b6 . . . b0. We'll term that byte the control byte.

We call such arrangements memory-mapped output, since what we put in memory maps, or sets the pattern for, what happens in the outside world. Most microprocessors support this or some similar sort of

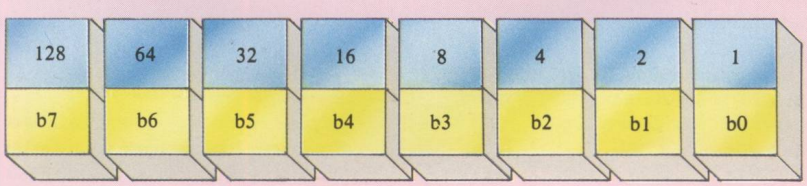


Figure I: Values associated with bit positions

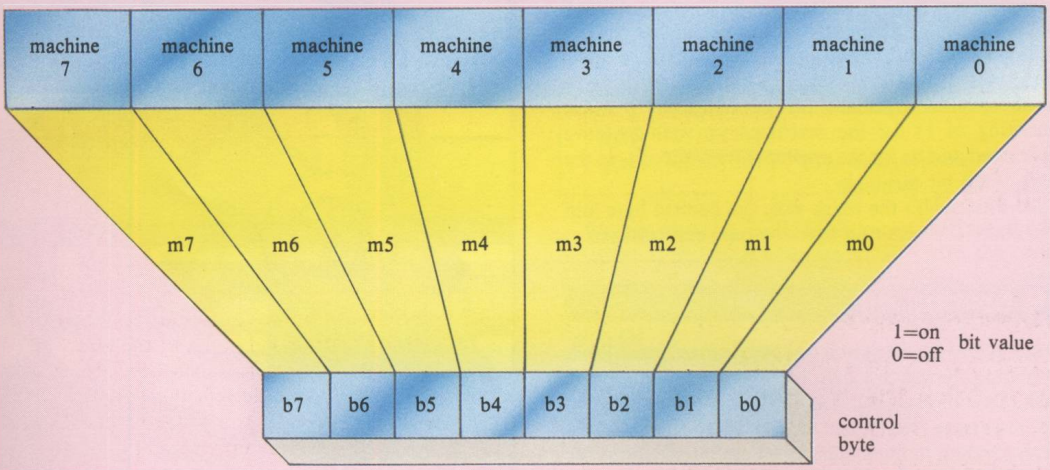


Figure II: Memory mapped control

output. Figure II shows the type of scheme we mean.

Assuming we've got things connected up properly, if we then load the control byte with:

%11111111

all the machines would be on. Remember that if a bit is set the corresponding machine is on. If we want to switch all the machines off, we can load the control byte with:

%00000000

And, of course, we can have any on/off pattern of machines, setting or clearing the relevant bits by loading the control byte with new numbers. Loading it with:

%11110000

is one way of switching off half the machines.

AND switching off

Sometimes, though, we might want to switch a particular machine or two on or off without knowing (or caring) whether the others are on or off. This means we need some way of affecting only the bits controlling those machines, while leaving the others unchanged.

Suppose we wanted to switch off a machine – say m6. We can do this by making b6 of the control byte zero. To clear that one bit to zero we AND the control byte with another byte – called the mask – the bits of which are set (1) except for b6, which will be 0. That is, we AND the control byte with:

%10111111

We then make this result our new control byte, and off the machine goes. To see how it works in practice, let's assume that initially all the machines are on, so the control byte is:

%11111111

To switch machine m6 off we must AND it with:

%1011111

The sum is:

%11111111 Control byte
AND %10111111 Mask
%10111111 New control Byte

As you can see, the outcome is that when we update the control byte with the result, m6 is switched off while the others remain on.

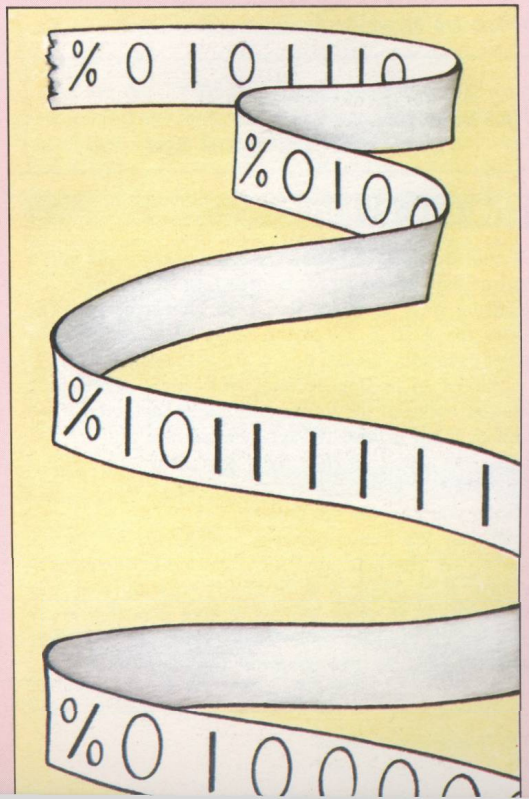
The trick isn't hard to see. Let's consider things from the point of view of bits in the mask. If the bit is a 1, when you AND it with the relevant control bit the resulting bit is the same as the control bit. That is, ANDing a bit with 1 leaves that bit unchanged.

Think about it. If the control bit were 1, then as 1 AND 1 = 1, you're left with 1. The bit's unchanged. If, on the other hand, the control bit were 0 then, as 0 AND 1 = 0, the bit remains unchanged as 0. In other words bits in the mask with 1 in them leave the corresponding control bit unchanged.

So for machines whose on/off status we don't want to alter – we may not even know if they're on or off – we set the corresponding bit in the mask to 1.

However if the bit in the mask were clear (0) it wouldn't matter what the state of the original control bit was – the result would still be 0.

Say the control bit was 1, then as 1 AND 0 = 0 the



resulting bit is a 0. Alternatively, if it were 0, since 0 AND 0 = 0 the resulting bit is again 0.

So bits in the mask with 0 in them set the corresponding bits in the result byte to 0. This means to switch specific machines off we construct a mask consisting of 1s for the machines we wish to leave unchanged and 0s for the machines we want off – in the appropriate bit positions.

We then AND the mask with the control byte and then make the resulting byte the new control byte.

OR turning on

Fine, but how do we switch on specific machines? Well, we update the control byte by ORing it with another mask. This time we put 1 in the bits corresponding to the machine we want on, and 0 in the bits corresponding to the machines whose on/off status we wish to leave unchanged.

This works, since when you OR a bit (whether 0 or 1) with another bit whose value is 1, the answer is 1. That is **0 OR 1 = 1** and **1 OR 1 = 1.**

So using a 1 in the relevant bit of an OR mask will set the corresponding result bit. When this becomes the new control byte the corresponding machine will be turned or left on.

On the other hand, ORing a bit in the control byte (no matter what value) with 0 leaves that bit totally unchanged since **1 OR 0 = 1** and **0 OR 0 = 0.**

So when we OR the bits of the mask that are 0 leave the corresponding bits of the control byte unchanged. This means, to switch specific machines on we use a mask consisting of 0s for the machines we wish to leave unchanged, and 1s for the machines we want on – in the appropriate bit positions.

We then OR that mask with the control byte and make the resulting byte the new control byte. Hence, to ensure that m6 is definitely on, we OR the control byte with:

%01000000

For example, if m6 is off, and all the rest on, to switch m6 on we do the following:

```
      %10111111 Control byte
AND   %01000000 Mask
      %11111111 New control Byte
```

The use of force

Of course, both AND and OR have uses for the micro enthusiast other than controlling machines. To illustrate one, consider the Ascii character set. The codes for A to Z are in the range 65-90, while their lower case equivalents, a to z, are in the range 97-122.

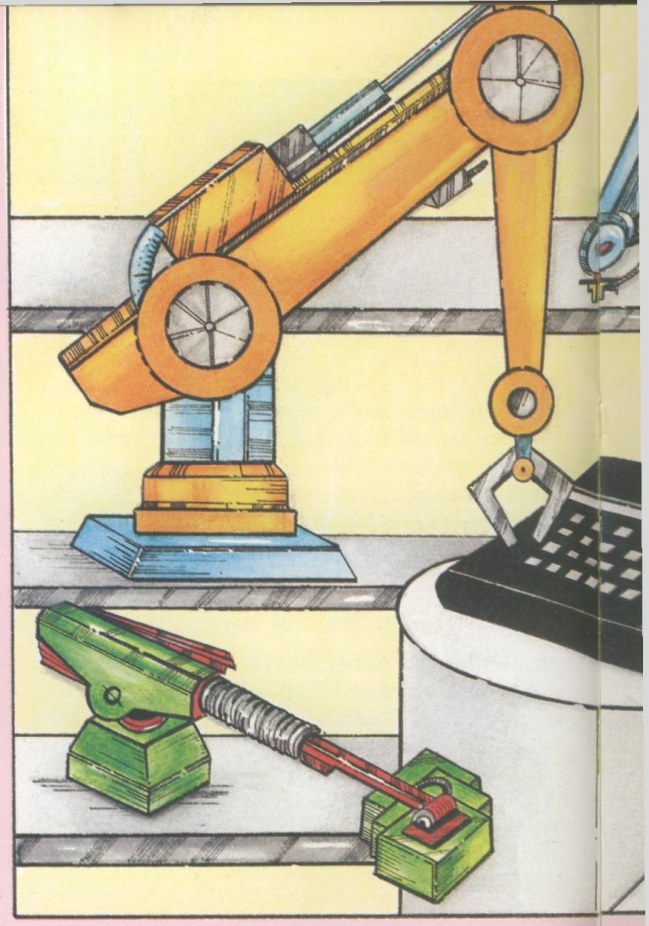
Looked at in this decimal way, there seems little relation between the upper and lower case sets. If we look at them in hex, though, we can see that:

A . . . Z runs from &41 to &5A
a . . . z runs from &61 to &7A

I hope you can see the pattern.

In fact the numerical Ascii difference between a lower case character and its upper case equivalent is always &20. Looked at in binary, this difference is %00100000. In other words, bit five is set for lower case, and is clear for upper case – remember, we start with the zero bit. For example, the code for A is:

%01000001



whereas the code for a is:

%01100001

Similarly, the code for Z is:

%01011010

and the code for z is:

%01111010

In both cases the only difference is in bit five. So if we have an Ascii code for a letter, we can *force* it to be upper case by clearing bit five to zero. We can do this by ANDing the code for the letter with the mask %11011111 (&DF).

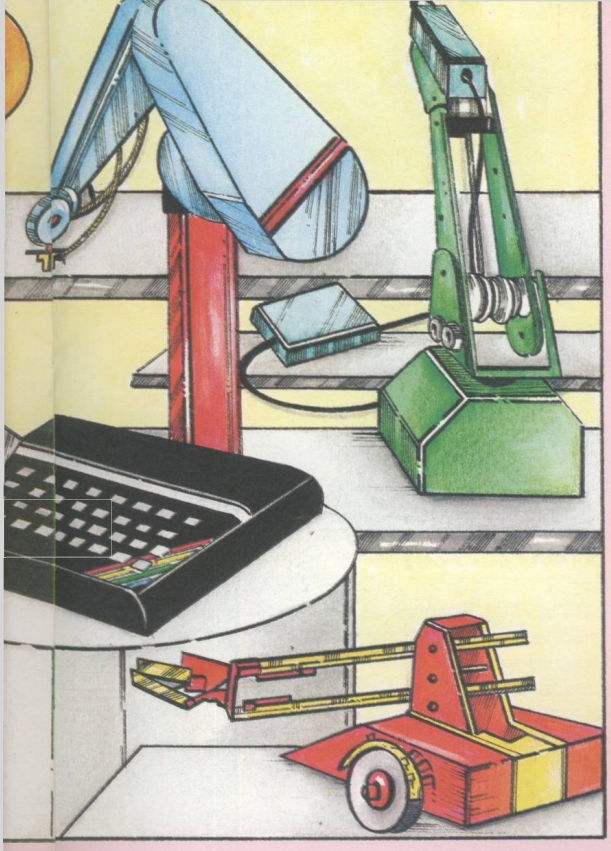
Remember, the bits in the mask that contain 1 will leave the corresponding bits in the Ascii code for the letter unchanged in the resultant byte, whether they be 0 or 1. On the other hand, the bit in the mask with 0 in it will force the matching result bit to be zero. So:

```
      %01100001 (The code for a)
AND   %11011111 (The mask – &DF)
gives %01000001 (The code for A)
```

It won't surprise you to learn that we can reverse the procedure – forcing upper case into lower case – by using OR to set bit five. This time the mask will be %00100000, the 0s leaving things unchanged in the resultant byte, the 1 forcing a corresponding 1 in bit five of the result bit. So:

```
      %01011010 (The code for Z)
OR    %00100000 (The mask – &20)
gives %01111010 (The code for z)
```

One further use for AND is to test if a particular bit in a byte is set. We just AND that byte with a mask



consisting of a 1 in the bit being tested, with 0s in all the rest. The bits with 0 in them, of course, set the corresponding bits in the resultant byte to zero.

Since the rest of the bits are already cleared to zero by the mask, the only thing that could stop the entire resultant byte being zero is the value derived from the bit under investigation:

- If that bit is set, the corresponding result bit will be set also (1 AND 1 = 1) so the resultant byte will be non-zero.
- If the bit being checked is clear, the corresponding result bit will be clear (0 AND 1 = 0) so the resultant byte is zero.

Let's see how this works in practice. If we were testing for bit four being set, the mask would be %00010000. Try ANDing this value with %00110100, where bit four is set, and also with %00101100, where bit four is clear, and you'll see that the resulting bytes are non-zero and zero respectively.

Hey presto, it's XOR!

So what of EOR/XOR? Well, its function is to return a 1 if the pair of bits being combined differ, and 0 if they're identical. Given this, we can use XOR to test which bytes in a bit differ. For example:

```
    %10101110
XOR %11001101
gives %01100011
```

where the set bits neatly mark out the differing pairs.

We can also use EOR/XOR to complement or NOT a byte, by XORing it with a mask of %11111111. Since the mask is all 1s, the result depends entirely on what's in the byte under investigation. Bits that contain 1s will give 0 (since 1 XOR 1 = 0), while bits that contain zero will give 1, since 0 XOR 1 = 1.

This is exactly what we want to happen with a NOT – change the 0s to 1s and vice versa. For example:

```
    %10101101
XOR %11111111
gives %01010010   (The complement)
```

We can also use EOR/XOR to test if two bytes are identical. If the result when we XOR is zero, they must have been identical since every pair of bits must have given zero, which only happens when the bit values are the same.

If there's a non-zero result there must have been a pair of bits that differ, so the two bytes under consideration must differ. For example:

```
    %10101010
XOR %10101010
gives %00000000
```

whereas:

```
    %10101110
XOR %10101010
gives %00000100
```

which is, of course, non-zero, since the bytes differ.

You might have come across EOR/XOR in graphics application programs where it's widely used for its "hey presto" effect. This is based on the fact that if you XOR a first byte with a second and then XOR the result of that once more with the second byte, the first byte reappears. Look at this, if you don't believe me:

```
    %01011100   (First byte)
XOR %01110010   (Second byte)
    %00101110   (Result)
XOR %01110010   (Second byte again)
    %01011100   (First byte back!)
```

We use this XORing technique to draw things on a background and then move on, leaving the background unchanged. In this case the first byte is the background pen number. If we then XOR our second byte – corresponding to the ink number of whatever it is we're drawing – on to the background, it will be displayed in the resultant ink number. It's rather like mixing colours mathematically.

To get rid of what we've drawn, we draw it again with the same ink number, once more under the influence of XOR. Of course XORing twice with the same byte gives us the original byte back. This results in whatever it is being drawn appearing in the original, background colour. Hey presto – it's gone!

That's it . . .

And that's the end of the series. Hopefully you'll have gained some idea of the power of binary numbers and the ways they can be combined. I've only touched on a fraction of the potential uses, but you'll be well equipped to work things out for yourself from now on.

And if you're looking to take these ideas further, why not try the machine code series we've featured? These articles should get you off to a flying start.

Into adventures



What . . . Why . . .

ADVENTURE games on the Spectrum are a natural extension of such role playing games as Dungeons and Dragons or even Cluedo. They give you, the player, freedom of movement and freedom of action within a set group of locations, within which you can wander around, effectively doing your own thing.

Imagine a vast movie set, dozens of actors and hundreds of props. You have read the draft script and know a little of what to expect as you move around. So you know that it is a detective story, or a swords and sorcery treasure hunt.

You are to play the lead part. You may move and act in whatever fashion you choose – this is the fascination and attraction of the adventure game. If you wish, you can ignore the plot and explore your new world just as the fancy takes you.

A good game, be it text-only or the most sophisticated graphic display imaginable, transports you to another reality. It creates an alternative universe where you really can believe you are leading a completely independent existence – where dragons still roam, or life depends upon finding that hidden cylinder of oxygen.

Not only can you move around at will but usually you can collect, examine and use the objects you see along the way, climb up cliffs, swim rivers, open doors, eat, drink and generally take on the persona of your other self.

The final objective will vary from game to game but all contain a series of puzzles that you have to solve.

PEET 86



Those that accept typed-in commands usually call for mental dexterity with no time limit, whereas those employing a joystick often call for a quick hand and eye.

Which . . .

Adventure games come with all types of plots and scenarios, from deep hidden dungeons to spaceships in the far future, from prehistoric dinosaurs to Victorian detection. There should be something to please everyone.

There are three main types of presentation – text-only, text-with-graphics and graphics-only. As with almost anything you buy, there are good and bad examples of each type, so either ask to see the program demonstrated or read its reviews in computer magazines. At least you will then have some idea of what you are buying.

Many people favour text-only games as these usually mean that a fair amount of descriptive text is displayed as you visit different locations. Also, your mind is far better at providing a detailed picture of your surroundings than the best graphics yet available on the Spectrum. If you think about it, very few classic novels have pictures.

Text-with-graphics programs vary considerably – some have good graphics and not much text, a very few have good graphics and good text. Many have mediocre graphics that take some time to draw and would be better as text-only.

Graphics-only adventures usually use a joystick to control the progress of the game and can be roughly divided into two groups. First, and largest, is the arcade adventures group where you guide your alter ego through different locations. Other actions – picking up, examining, shooting and so on – are achieved by certain joystick combinations or by use of the keyboard.

Secondly there are those games that employ a series of icons or small pictures, over which you move a cursor to select a command. In addition to the picture of your location will be icons representing the objects lying around and those carried, together with those that stand for permissible actions such as get, drop and open.

To actually do something, you guide the cursor first over the action icon, then the object icon. Although this technique is quite impressive and is ideal for players who cannot read or spell, it can sometimes take longer to give and select a command than to type in the appropriate words.



How . . .

You have loaded your first adventure and the instructions have given you some idea of your final objective. So how do you start? Most people, even hardened adventurers, charge around a bit to see what is there!

Having got that out of your system, you can now start playing properly. It is wise to start from scratch, as the number of moves you have made may have been counted. This could mean that it is near moonrise or, if you have a lamp, it could very soon run out of oil or batteries. Programmers are devious creatures and often want to keep you in the dark, literally.

Most games will allow you to quit, so type in QUIT and press Enter. More often than not you will be asked if you wish to start again. Alternatively you may have to switch everything off and reload the program.

The first thing to remember is that you must not get



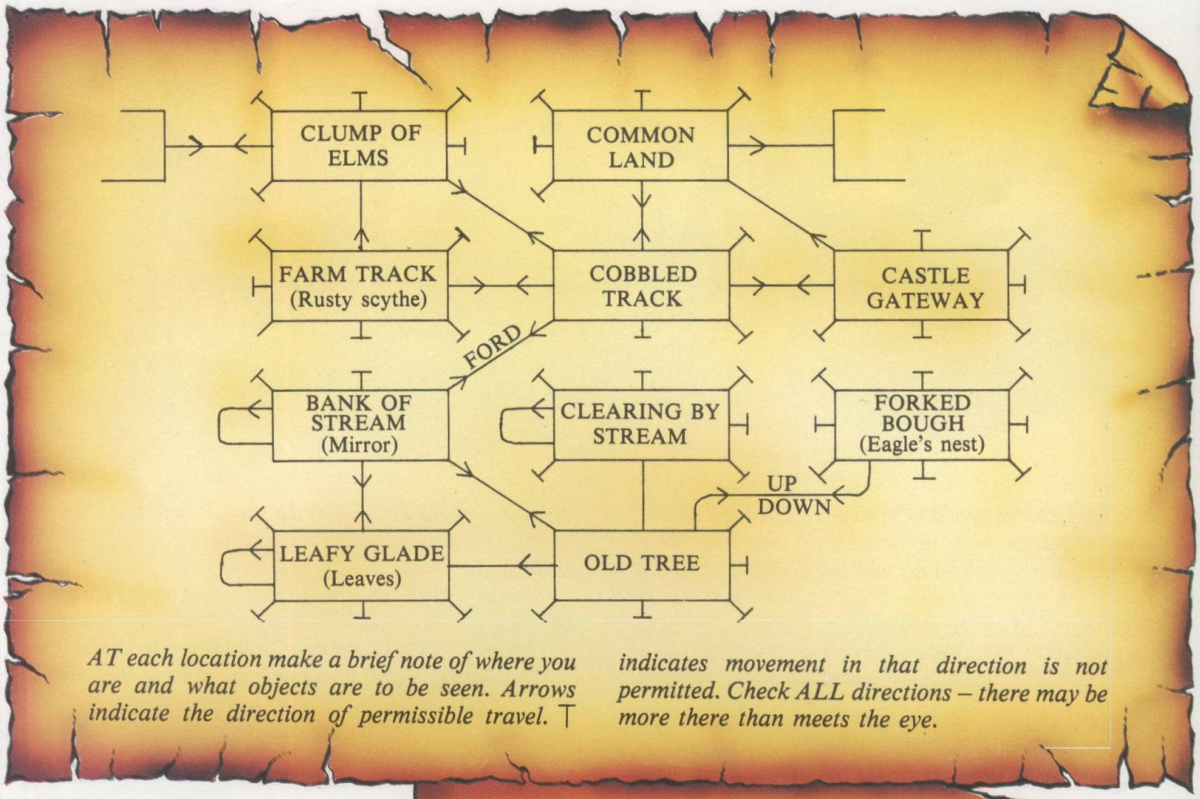
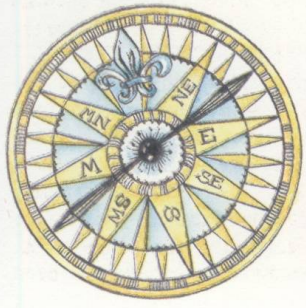


Figure I: General mapping



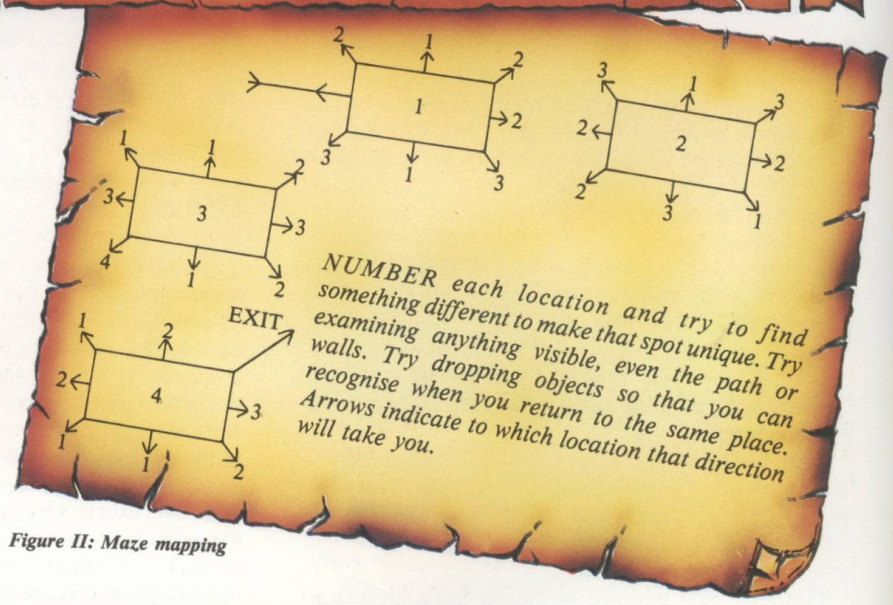


Figure II: Maze mapping

lost. Right from the start you should get into the habit of drawing maps. These need only be a series of boxes, with the location written in them. They should be linked by lines (North, East, South, West) to show the direction of one location from another. Note where you can only travel in one direction. Figure I shows you the idea.

Not only will this help you to know where you are but it will also give you a good idea where you have not yet been. Exploration and discovery are what adventures are all about, so go everywhere and examine everything.

Some games allow you to EXAMINE objects before you have picked them up. Find out which type you are playing and have a good look at everything you can – you will be surprised at what you will find. Objects to be found will fall roughly into three categories – treasure, useful, and red herrings. The first are usually pretty obvious but the latter two can only be decided by playing the game.

A common problem is to find that you can only explore a limited number of locations because something or someone inhibits your progress. Perhaps it is as simple as a locked door, where you have to find the right key, or you may need some rope to climb out of a window.

The puzzles may seem difficult, but as you play you will find there is often a pattern or theme, similar to crossword puzzles in different newspapers. Also, just like a good crossword, having solved the clue you'll kick yourself for not seeing the answer sooner.

When . . .

Not only must you learn how to move around freely but you also have to find the correct order in which the puzzles have to be solved. It is no good parachuting from an unclimbable cliff, only to find you do not have

Text adventures – useful words

HELP	Type this and see what happens – many games will give you a cryptic clue that may be useful.
INVENTORY	This will show you what you are carrying – try it at the beginning of a game, you may find you already have some useful items.
LOOK	Will usually cause a repeat of the present location description – useful if the text scrolls and you have forgotten details of exits and so on.
SAVE	Enables you to SAVE your game position.
LOAD	Enables you to LOAD in the data from a previously SAVEd game. Some games use RESTORE.
EXAMINE	Gives you further information about specified object, also try SEARCH. If the object could conceivably be read, try READ.
QUIT	Usually allows you to start again at the beginning of an adventure without having to load in the program from tape.
WAIT	In some games a useful command to allow subjective time to pass while waiting for something to happen.
GET	Sometimes TAKE – for picking things up, perhaps to EXAMINE more closely.
DROP	Sometimes PUT – for putting down what you are carrying. Beware of dropping things breakable.



the magic sword that defeats the dragon waiting round the corner!

Sometimes the future use for an object is obvious, all too often it is not. The amount you can carry is usually limited to only a few items, so you cannot solve the problem by carrying everything.

Fortunately all the better adventures allow you to SAVE your position – together with all you are carrying – on to a blank tape of your own. This will allow you to carry on from your last SAVEd position when catastrophe occurs. Also, if you SAVE your position regularly as you proceed it will enable you to backtrack when you find you have missed some vital clue.

If the adventure is divided into obvious compartments make sure you have thoroughly investigated every location before moving on to the next phase. It is fairly common practice to find that an apparently useless item in an early part of the game is vital later on. Likewise some problems set in early phases need something found later to solve them. You don't have to have a devious mind to play adventures but it helps!

Where . . .

In many games only the cardinal points are recognised – North, South, East and West – but beware of the occasional South East or North West. When you are mapping the terrain always try these directions – you never know what may happen! Remember that UP and DOWN are also movement commands – you may even find that in a forest CLIMB TREE has no effect but UP does.

Although many words will be recognised by the program, not all games have extensive vocabularies. This is not necessarily the sign of a poor game but it does mean you have to learn how to use what is there.

Wanting to enter a hut with a door to your north could involve one of the following: OPEN DOOR then GO NORTH, IN, GO DOOR, GO HUT, ENTER or ENTER HUT, or just plain NORTH. You must learn how to interact with that specific game.

This learning process is usually quite quick and you will often find help by reading the instructions first. Icon driven games do not have this problem. What is possible is clearly defined by movement icons shown on the screen. This is a mixed blessing as sometimes that feeling of freedom of action is lost.

There is one other movement problem found in nearly every adventure game – mazes. You will find yourself in a series of locations that all appear the same. Your careful mapping will now no longer tell you where you are, because moving West after moving East will not bring you back to where you started. You must map these mazes carefully and completely. They often hide items you need or the way on to another series of locations.

The traditional way of solving a maze is to drop an item at each location, so making the description of that place unique. Instead of drawing a series of boxes linked by lines, just draw the boxes – each numbered – with arrows showing where that direction leads to. Figure II shows you an example.

Logically you should be able to map a six location maze with five dropped objects. Of course programmers are aware of this, so they are forever trying to think up more devious mazes. You may find an eight location maze and only be able to carry four objects.

Perhaps you could go back and collect more things to drop or maybe part of the maze can be mapped before entering the other part. Try examining everything – you may find something hidden that will define a separate location. Look very carefully at either the graphics or the text, perhaps there is a minor difference you did not spot at first.

The permutations for adventure games seem endless. Spells may be cast, passing strangers may be talked to, or computers may be quizzed. Alternate universes are out there just waiting for another intrepid adventurer . . . like yourself.

Creating adventures

Take the challenge one step further by actually writing your own adventure game

IF you want to write adventures, there are really only two ways you can go. One is to use an adventure creation program and the other is to go the whole hog and write the entire program yourself. But this will take time and a great deal of thought – even before you type in anything at all.

Writing an adventure is like playing one. There are puzzles you will have to solve and your mapping of the game must be accurate. It is all too easy to write one that is almost impossible to solve – the knack is producing something that is just difficult enough to make the player want to continue to the end.

Puzzles are what make a game, together with the feeling that the computer has some idea of what the player is trying to achieve. There is nothing worse than the bald reply of "You can't". Think up humorous or ambiguous replies that give the impression the computer knows exactly what is wanted but chooses not to comply for some reason.

The plot need not be developed in full but you must have some fairly concrete ideas on what the player will have to do to complete the game. Having got more than a glimmer of an idea, start drawing out a map of the terrain. Draw a number of boxes in a grid pattern and number each box. These represent locations and should be large enough to write where it is and what objects – if any – will be found there. Boxes should connect in the main compass directions but do not have to connect in all directions.

Having got your basic plot and map sorted out, you must refine them to form a logical and consistent whole. You will have thought of several good puzzles for your adventure and having drawn the map will have included the objects you wish to use. Now is the time to consider whether there are enough puzzles to hold the player's attention and whether the objects to be found are plausible at that location.

You are creating a world of your own, so you can dictate the logic of that world – but it must seem reasonable to anyone playing the game. Finding a silver sword just lying on a path may not be so acceptable as finding it hidden behind a stone slab. Whatever you do, do not jar a player's acceptance of what is seen on the screen.

On your own

Writing your own adventures from scratch is a very satisfying and challenging project. It can also provide a very adaptable operating system or core program, providing you are prepared to spend time refining what you learn after writing each adventure.

There are many ways of programming adventures and the mini-adventure shown here, Gravely Manor, uses a very simple direct method. It works well enough but is without any special frills. Used as it stands it will provide the bones on which to build a more elegant system of your own.

First consider what is happening when you play an

The Quill – Gilsoft

This is the only adventure creation program available for the Spectrum and is not difficult to use. It seems complex at first but don't panic, all will come into focus as you progress. Having briefly outlined some of the functions of the program, you are given step by step instructions on creating a simple adventure. Follow these and The Quill will begin to come alive.

Although any adventure must be planned carefully in advance, The Quill has excellent editing facilities, so alteration of locations, objects, vocabulary, messages and even directions can all be made with reasonable ease. Sound may be incorporated using the Spectrum's BEEP command and user defined graphics (UDGs) may also be handled, enabling you to have rudimentary graphics.

Each area of information can be accessed from the main menu and then individual data about one particular location or object entered or altered. The program even checks that the entry is valid with respect to the other entries already made. It will not permit you to refer to a location or object that does not exist.

There is no doubt that The Quill provides the most painless way of creating your own adventures. Several programs created with its help have become best sellers and Gilsoft request only that The Quill be credited somewhere within the program.

Taking note of a growing demand for adventures with graphics, Gilsoft introduced The Illustrator. This enables pictures to be added to programs written with The Quill. Even so, creating good graphics is not only time consuming but also requires a certain artistic flair.

Pictures must be very carefully sketched out on graph paper before they are translated into commands that The Illustrator will act on. This requires a fair degree of dedication but the result can enhance the game's attraction enormously. You would be well advised to become adept at using The Quill before adding any pictures.

After The Illustrator came The Patch, offering

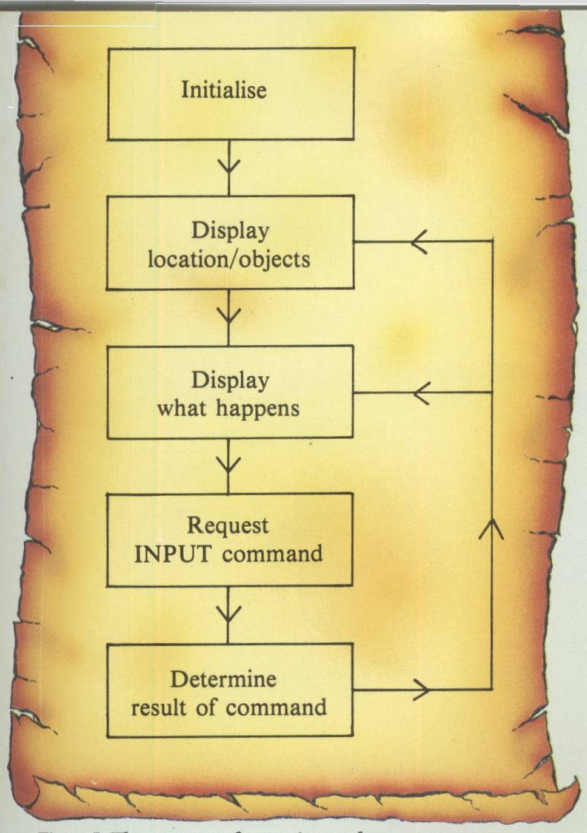


Figure I: The sequence of events in an adventure program

adventure. Figure I shows a simple diagram of what happens. Once you load and run the program, the first thing is to set up such things as variables, arrays and perhaps display instructions. You then enter a loop that will only be broken out of if the player wins, dies or quits.

This starts with the display of the first location, any objects visible and perhaps the directions in which you can move. The program then waits for you to enter a command of some sort. This could be a direction in which to move, or some other action command.

Whatever you type in will produce some response, even if it is an ambiguous answer meaning that the game does not have anything of importance to tell. If the command requires a move to a new location then you are back at the beginning of the loop. This procedure is repeated until you win, lose or quit.

As a programmer you will have to look at this loop more closely and begin to think how to code the various sections. Gravely Manor uses a similar construction, so you will be able to harvest some of the routines for your own games.

The program is directed to the initialisation routine (8000) at line 10. This is at the end as it is only accessed once and could have a substantial number of lines. All GOTOs and GOSUBs are found by the computer counting from line 1, extra lines at the beginning would tend to slow down the response to input commands.

This game only has nine locations and all the information for these is in lines 8101 to 8209. The DATA in 8101 is READ at 8035 into six arrays N(),E()...D(). This defines the location reached in moving in any of these six directions from the present location. So, the first six numbers will appear in N(1), E(1) and so on. This allows exits only to the North – to location 2 – and the South – to location 10 which is a special case and only brings you back to location 1 after a message.

If writing your own game, only type in six direction numbers on each line. This will enable you to check your map more easily. They are all on one line here to make it more difficult to cheat if playing the game.

So that the game is not too easy to play, no REMs are shown but to gain from it, you must understand what is happening. After playing it, you can learn a great deal by inserting your own REMs. Here are some line numbers to get you started:

99 Display inventory
399 Start game loop
499 Print location and objects
599 Input and parser
699 Check for verb
799 Single key/word inputs
899 Check for object
999 Direct program for action

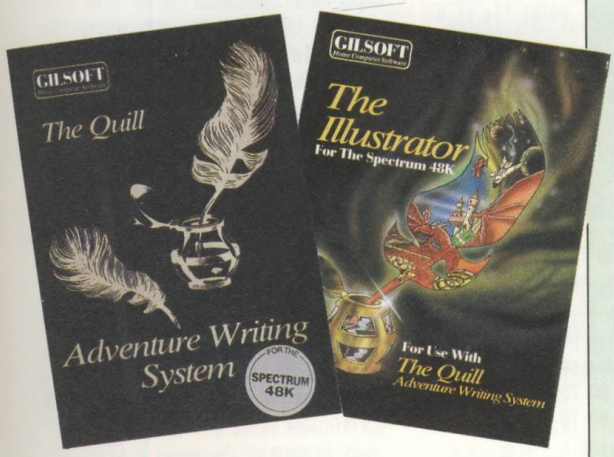
If you study the lines following 699 and 899, you will see that they check the input against the lists of verbs and objects in *V$* and *N$* (8015, 8020). This will give values V and N that enable the program to be directed to the appropriate action and response in 1010. 18 is subtracted from V because verbs below that number have already been dealt with as single key or word entries such as N, EAST or INVENTORY.

To study these actions and responses you will have to work out the REMs yourself, starting with 1099 CLIMB. EXAMINE is an important verb and will direct the program to a series of lines starting at 4000 for each object recognised in *N$*.

The object DATA (8301-8309) is put into three arrays *P(), C()* and *O$()*. The first defines the object's initial location and the second is its class. Class is an important attribute, and may be used in many ways. It is used here to indicate whether the object is immediately visible (505) and its weight (1510).

Only a brief explanation is possible in such a short space but once hooked, you will find writing adventure games just as fascinating as playing them. There are many similarities and more ways of constructing them than you would first imagine. The more you write, the more you will learn about the workings of the Spectrum.

Now turn the page and discover the mysteries of Gravely Manor.



additional facilities that may be combined with the other two programs. They include an excellent quicksave/load routine to and from RAM and the ability to alter repetitive system messages.

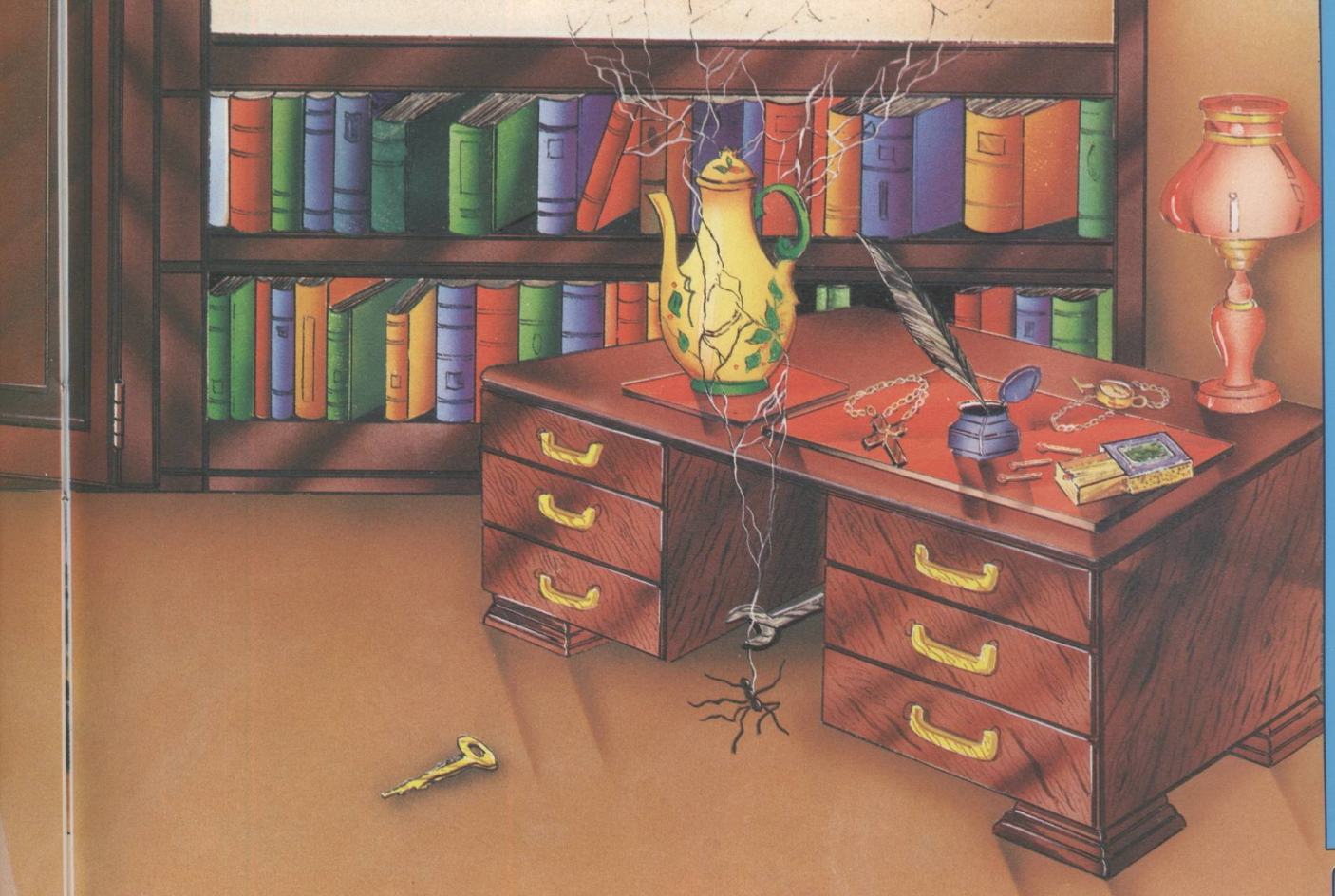
The three programs from Gilsoft are a complete package and once mastered give you the opportunity of creating top adventure games.

Gravely Manor



Unravel the secrets of Gravely Manor in search of a mysterious amulet

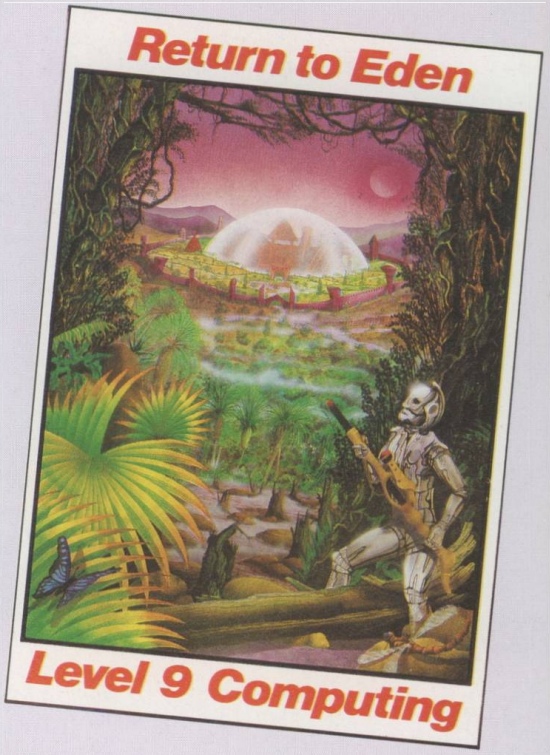
```
 10 GO TO 8000
 100 LET F=0: PRINT "You have:"
 105 FOR I=1 TO 11
 110 IF P(I)>0 THEN  GO TO 120
 115 LET F=F+1: PRINT " ";O$(I)
 120 NEXT I: IF F=0 THEN  PRINT "Noth
ing"
 125 GO TO E
 200 PRINT "Try something else!": GO
TO E
 300 PAUSE 60: GO TO D
 305 PAUSE 350: GO TO D
 400 CLS : IF L=10 THEN  PRINT "Mists
 swirl...": LET L=1: GO TO B
 405 IF L>7 AND (P<3 OR P(5)>0) THEN
 PRINT "It's very dark!": GO TO E
 450 PRINT L$(L)
 500 LET F=0: PRINT P$;"You see:"
 505 FOR I=1 TO 10: IF P(I)<>L OR C(I
)>3 THEN  GO TO 515
 510 PRINT " "+O$(I): LET F=F+1
 515 NEXT I: PRINT
 520 IF F=0 THEN  PRINT "Not much"
 600 POKE K,0: PRINT : INPUT "What no
w? "; LINE I$: IF I$="" THEN  GO TO E
 605 LET I$=I$+"  ": LET Y=1: LET V=0
: LET N=0: LET W=0: LET W$(1)="": LET
 W$(2)=""
 610 FOR J=1 TO 2
 615 FOR I=Y TO LEN I$
 620 IF I$(I)=" " THEN  LET W$(J)=I$(
y TO i): LET Y=I+1: LET I=LEN I$
 625 NEXT I
 630 IF (W$(J)="the " OR W$(J)="a   "
 OR W$(J,1)=" ") AND Y<=LEN I$  THEN
 LET W$(J)="    ": GO TO 615
 635 IF W$(J,1)<>" "  THEN  LET W=W+1
 640 NEXT J
 700 FOR I=1 TO 108 STEP 4
 705 IF W$(1)=V$(I TO I+3) THEN  LET
V=1+(I-1)/4: LET I=108
 710 NEXT I
 715 IF V=0 THEN  GO TO A
 720 IF W>1 THEN  GO TO 900
 725 IF V>17 THEN  GO TO A
 730  LET I$=I$(1)
 800 IF I$="n" THEN  LET NL=N(L)
 805 IF I$="e" THEN  LET NL=E(L)
 810 IF I$="s" THEN  LET NL=S(L)
 815 IF I$="w" THEN  LET NL=W(L)
 820 IF I$="u" THEN  LET NL=U(L)
 825 IF I$="d" THEN  LET NL=D(L)
 830 IF I$="i" THEN  GO TO 100
 835 IF I$="l" THEN  GO TO D
 840 IF I$="q" THEN  GO TO 2100
 845 IF NL=0 THEN  PRINT "You cannot
go that way": GO TO E
 850  LET L=NL: GO TO D
 900  FOR I=1 TO 112 STEP 4
 905 IF W$(2)=N$(I TO I+3) THEN  LET
N=1+(I-1)/4: LET I=112
 910 NEXT I: IF N=0 THEN  GO TO A
1000 IF V=23 AND N>16 THEN  LET I$=W$
(2,1): GO TO 800
1005 IF N<10 THEN  IF (P(N)<>L AND P(
N)<>0) OR C(N)>5 THEN  PRINT "Is it h
ere?": GO TO E
1010  LET V=V-18: GO TO 1100+(V*100)
1100 IF L=6 AND (N=10 OR N=21 OR N=27
) THEN  LET L=5: GO TO D
1105 IF L=5 AND (N=10 OR N=22 OR N=28
) THEN  LET L=6: GO TO D
1110 PRINT "There is nowhere to climb
!": GO TO E
1200 IF N>9 THEN  GO TO A
1205 IF P(N)>0 THEN  PRINT "You are n
ot carrying that!": GO TO E
1210 IF N=5 AND P=1 THEN  LET P=0
1215 PRINT " OK...dropped": LET P(N)=
L: LET WT=WT-C(N): GO TO B
1300 IF N>16 THEN  GO TO A
1305 GO TO 4000+(n*100)
1400 IF (P(1)<1 OR P(1)=L) AND P>1 TH
EN  PRINT "Its already full!": GO TO
E
1405 IF P(1)<1 OR P(1)=L THEN  PRINT
"You fill the lamp with oil...": LET
P=2: GO TO E
1410 GO TO A
1500 IF N>9 THEN  GO TO A
1505 IF P(N)<1 THEN  PRINT "You've al
ready got THAT!": GO TO E
1510 IF WT+C(N)>5 THEN  PRINT "You ca
nnot manage that": GO TO E
1515 IF N=5 AND P=0 THEN  LET P=1
1520 IF N=7 THEN  LET R=1
1525 PRINT " OK...carried": LET P(N)=
0: LET WT=WT+C(N): GO TO B
1600 PRINT "You can only 'go' NSEWU o
r D": GO TO E
1700 IF L<>3 OR R=0 THEN  GO TO A
1705 IF N=4 THEN  GO TO 9000
1710 IF N=9 THEN  GO TO 9100
1715 GO TO A
1800 IF N=6 THEN  PRINT "Be careful n
ot to waste them!": GO TO E
1805 IF N=5 AND P=1 THEN  PRINT "You
cannot. It's empty!": GO TO E
1810 IF N=5 AND P>1 THEN  PRINT "It b
urns brightly": LET P=3: GO TO B
1815 PRINT "Arsonist!": GO TO B
1900 IF L=4 AND N=11 THEN  PRINT "You
 struggle at the heavy desk  and noti
ce a lever": LET L$(4,93 TO )="By  th
e desk is a lever.": LET S=1: GO TO C
1905 IF L=2 AND N=12 AND P(8)>0 THEN
 PRINT "With bare hands?": GO TO B
1910 IF L=2 AND N=12 AND C(9)>1 AND P
(8)<1 THEN  PRINT "You lever the loos
e flag up withthe pickaxe and see som
ething": LET C(9)=1: GO TO C
1915 GO TO A
2000 IF L=4 AND N=13 AND S=1 THEN  PR
INT "You hear á low rumbling sound":
LET L$(6,97 TO )="A hidden doorway to
 the west hasstairs leading down.": L
ET D(6)=8: GO TO E
2005 GO TO A
2100 INPUT " Play again ? ";I$
2105 IF I$(1)="y" THEN  RUN
2110 STOP
4100 PRINT "Full of oil!": LET C(1)=3
: GO TO E
4200 PRINT "A history of Gravely Mano
r. Somepages speak of the first owner
'sAmulet of Power. It's claimed  tha
t it is hidden in the manor": GO TO E
```



```
4300 IF C(5)<2 THEN  PRINT O$(3): GO
TO E
4305 IF L=1 THEN  PRINT "You see some
thing hidden": LET C(5)=1: GO TO B
4400  PRINT O$(4): GO TO E
4500 IF P<3 THEN  PRINT O$(5): GO TO
E
4505 IF P=3 THEN  PRINT "The lamp bur
ns steadily": GO TO E
4600 PRINT "Safety type": GO TO E
4700 PRINT "An old gentleman with an
amulet around his neck": GO TO E
4800 PRINT "Must have been a strong m
an who used this!": GO TO E
4900 PRINT "Carved in a most intricat
e      toothed pattern": GO TO E
5000 IF L=6 OR L=5 THEN  PRINT "What
a size, a sweeps delight!": GO TO E
5005 GO TO A
5100 IF L<>4 THEN  GO TO A
5105 PRINT "Looks heavy": GO TO E
5200 IF L=3 THEN  PRINT "Heavy local
stone": GO TO E
5205 IF L=2 THEN  PRINT "One looks as
 though it is loose": GO TO E
5210 GO TO A
5300 IF L<>4 THEN  GO TO A
5305 PRINT "Looks stiff!": GO TO E
5400 IF L=3 AND R=1 THEN  PRINT "That
 could be a keyhole": GO TO E
5405 IF L=2 OR L=3 THEN  PRINT "Grubb
y but still sound": GO TO E
5410 GO TO A
5500 IF L<>9 OR C(4)=1 THEN  GO TO A
5505 PRINT "Something lies in the dus
t": LET C(4)=1: LET C(8)=3: GO TO B
5600  IF L=2 OR L=3 THEN  GO TO 5400
5605 PRINT "Dusty walls": GO TO E
8000 DIM W$(2,4): DIM L$(9,150): DIM
O$(11,26): DIM N(9): DIM E(9): DIM S(
9): DIM W(9): DIM U(9): DIM D(9): DIM
 P(11): DIM C(11)
8005  LET P$=CHR$ 13: LET I$=""
8010 LET A=200: LET B=300: LET C=305:
 LET D=400: LET E=600: LET F=0: LET L
=1: LET NL=0: LET P=0: LET R=0: LET S
=0: LET WT=1: LET K=23658
8015  LET v$="n  e  s  w  u  d
 i  l  norteastsoutwestup  downinve
lookquitcliadropexamfillget go  insel
ighmovepull"
8020 LET n$="barrbookcobwkey lampmatc
painpickrod chimadeskflaglevepaneshelw
alln  e  s  w  u  d  norteastso
utwestup  down"
8025 PRINT P$;P$;"          GRAVELY MA
NOR";P$;"You are lost in mist and stu
mbleacross a derelict house. Solve  i
ts riddle or stay for ever...  "
8030 RESTORE 8101: FOR I=1 TO 9
8035 READ N(I),E(I),S(I),W(I),U(I),D(
I): NEXT I
8040  RESTORE 8201: FOR I=1 TO 9
8045 READ L$(I): NEXT I
8050 RESTORE 8301: FOR I=1 TO 9
8055 READ P(I),C(I),O$(I): NEXT I
8060 GO TO C
8101 DATA 2,0,10,0,0,0,3,0,1,7,0,0,4,
0,2,6,0,0,0,0,3,0,0,0,0,0,0,0,0,6,0,3
,7,0,5,0,6,2,0,0,0,0,0,0,9,0,6,0,8,0,
0,0,0,0
8201 DATA "Entrance to a derelict old
 manorhouse. The walls are covered
 with festoons of cobwebs."
8202 DATA "South end of the Great Hal
l, theancient flagstones are broken.
 The walls are panelled in oak."
8203 DATA "The north end of the Great
 Hall.The floor is smoother and the
 atmosphere lighter."
8204 DATA "The library has suffered w
ith  the passing centuries. A desk
 still stands in one corner."
8205 DATA "A dimly lit old storeroom
above the kitchen. Dust lies over som
ebarrels lying in the corner."
8206 DATA "The manor kitchen. Chains
hang  down from hooks in the chimney,
 probably for a large cauldron."
8207 DATA "Cool, airy room with windo
ws.It was probably a serving room."
8208 DATA "The kitchen cellars, compl
etely empty except for a covering of
 thick dust."
8209 DATA "A long, low stone room wit
h manystone shelves around the walls.
"
8301 DATA 5,4,"A large barrel"
8302 DATA 4,1,"An ancient book"
8303 DATA 1,4,"A sticky cobweb"
8304 DATA 9,6,"A large metal key"
8305 DATA 1,6,"A handsome oil lamp"
8306 DATA 0,1,"Box of matches"
8307 DATA 3,3,"An old painting"
8308 DATA 9,6,"A rusty pickaxe"
8309 DATA 2,6,"A bright metal rod"
9000 PRINT "The blades of a strange d
evice  spring from a panel and you ar
e cut to shreds. It seems you musthav
e made a fatal mistake!": GO TO 2100
9100 CLS : PRINT P$;"You have a visio
n of the amulet seen in the painting
and a voicethanks you for recovering
it.   Mists swirl and you are back in
 the sunny hills near your home."
9105 GO TO 2100
```

Something for everyone

An adventure can take on many forms, so there's sure to be one that will appeal to your taste



GIVE 10 people a selection of books or pictures and ask them to arrange them in order of preference and it's very unlikely they will all make their selections in the same order.

It's the same for adventure games. Some like a little humour, some prefer long descriptive text, some want the challenge of difficult puzzles and yet others want the freedom to roam around many different locations.

If the number of games used for such a test were reasonably large, you would see a sifting upwards of a range of games that had caught the attention of most of our players. They may not be selected in the same order but the top 20 or so in each person's selection would contain many similar titles.

Top 10 charts are not always indicative of good games as they are mostly based on volume of sales and initially this does not always reflect on how good the game is. They are often influenced by the amount of advertising a software company has put into a product before its release. Also as book publishers have found out, a good cover can sell well even if the content is below par.

Fortunately – unlike books – reviews of computer games are more widely read. Find a reviewer with the same tastes as you and read his comments carefully before deciding what to buy next.

Right from the Spectrum's early days adventure games have proved to be a major part of its software diet. Recently a list was published containing more than 150 titles. How do you narrow this down to what may be considered a classic collection?

Obviously it must be shaded by the players' preference for type of adventure. This may be swords and sorcery, detective, science fiction, fantasy, medieval, modern, based on a book, or just plain zany. Some will have graphics and some not. Graphics will usually make for a more attractive presentation but rarely alter the playability of the game.

One of the first

No collection of adventure games would be complete without a version of the game considered to be the original – Adventure. This was written by Will Crowther in 1975 and subsequently enhanced by Don Woods. It was played on many mainframe computers in America and doubtless retarded the progress of science by many man-years!

As the memory capacity of home computers grew and programmers became more adept at squeezing gallons into pint pots, the Crowther and Woods adventure appeared on most home micros. There are several versions for the Spectrum. All are text only but, if you can find a copy, one of the earliest by Abersoft called Adventure 1 is a good implementation of the original.

There are two others that can be recommended – Classic Adventure from Melbourne House and Colossal Adventure from Level 9. The latter has an excellent end-game added to the original version, reputedly because Level 9 had advertised a 200 room adventure and then realised the original did not have enough locations.

These adventures take place in a series of underground caves where you attempt to find various treasures. Needless to say all is not plain sailing, as other creatures are determined to either steal the hard won treasure, or impede your progress to other interesting parts of this subterranean complex.

You give commands as two word inputs in the form verb/noun and many of our accepted adventure puzzles can trace their history back to this classic adventure.

Another program that has to be considered a classic and a worthwhile addition to anyone's collection is The Hobbit from Melbourne House, This was the first to feature graphics with real picture quality. It also scored a first by having independent characters roving around within the adventure that you could interact with meaningfully.

Although not the first game to be based on a well-known book it was certainly the first to be sold packaged complete with book. You play the part of

hobbit Bilbo Baggins who, with 13 dwarves and the wizard Gandalf, sets out on a quest for the Lonely Mountain. The journey has many dangers – trolls, goblins, elves and finally the dragon Smaug must be outwitted to achieve success. The game differs from the book and the slight variations make this adventure totally addictive.

Friends of the sword

There are many other games that fall within the realms of fantasy, so if the thought of venturing forth armed with your trusty sword appeals to you, then be prepared for a long and rich feast. Level 9 has several adventures in the mould of Colossal Adventure such as Adventure Quest and Dungeon Adventure. These have lots of amusing puzzles and many locations to explore. Both are text only games but produce such highly descriptive text that you have to question the present vogue of graphics with everything.

Three more text-only games come from Incentive Software, an interlinked trilogy – although each may be played separately – where your task is to search and destroy the evil sorcerer Vran Verusbel. Mountains of Ket, Temple of Vran and The Final Mission will each provide you with a series of brain gelling puzzles that are both logical and fiendishly simple – once you have solved them!

Melbourne House has now released a follow-up to The Hobbit, the first of three programs based on that classic trilogy The Lord of the Rings. It has very rudimentary graphics, several flaws that cause the program to crash and is very, very slow in operation. But if you are one of the many who have read and enjoyed the original books, you will find this adventure brings back all the tension and magic of Middle Earth.

Still in the realm of fantasy but with the added dimension of using magic spells, is Red Moon from Level 9. As we have come to expect from this company, there are some 200 locations, excellent descriptive text and a really workable combat and magic system. Red Moon also has very acceptable graphics and is probably somewhat easier than Level 9's other games. It's a good one for the beginner and has its fair share of Level 9 humour.

All the offerings from Adventure International are strong on fantasy, such as Robin of Sherwood, Gremlins, Spiderman and The Hulk. All have very good graphics and devious puzzles and although perhaps not the best games for a beginner, they will grow on you. No collection is complete without at least one from this stable.

One of the attractions of adventure games is the freedom of choice you have in your actions. One legitimate restriction may be on movement, as sometimes you need to solve a certain puzzle before another series of locations is opened up for exploration. All the adventures mentioned so far come into this category but if you really want to do your own thing then look more closely at some of the adventures from Beyond Software.

Adventure plus

1984 saw the introduction of Lords of Midnight, a mixture of both strategy and adventure quest. This has some 4000 locations and a graphic representation of what you would see if you looked in eight directions from each location – yes, some 32,000 views of the landscape! The aim is to defeat the evil Doomdark, who controls the north and whose forces are moving steadily southwards to engulf the remaining free lands.

You initially control the movement of four main characters and guide them wherever you choose. There are many other characters you may meet and try to recruit to your cause. Once they have joined you, you then control each of these independently. Each of the four main characters has different attributes and can best be guided to specific tasks – but the choice is yours.

Do you rally together thousands of loyal troops for major set battles or infiltrate the enemy's lines and strike for his source of strength? There is more than one way to defeat Doomdark, for this is a game with many possible outcomes and some of them are not what you would wish.

Beyond Software followed this with a sequel, Doomdark's Revenge, with even more possible views of the lands of Midnight – 48000. This is a complicated game also involving the raising of armies and special quests but is not recommended for novice adventurers who would be advised to sharpen their teeth on Lords of Midnight.

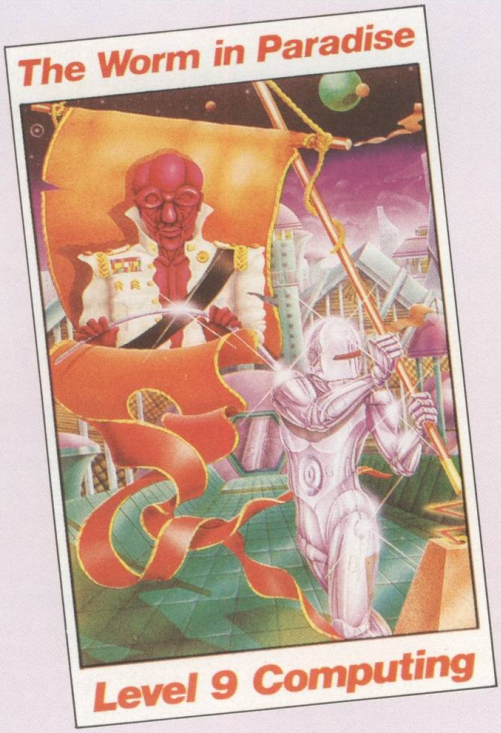
For those not so interested in the strategy of directing battles involving thousands of troops, Beyond has produced Sorderon's Shadow. This combines traditional text input with the landscaping effects used so successfully on its two previous strategy adventures.

This is role playing par excellence. There are many independent characters – good and bad – and a series of nine interlinked quests that are bound to keep even hardened adventurers on their toes.

Using not dissimilar techniques Firebird's program Runestone is another landscape you can walk into with the help of your trusty Spectrum. Three different characters may be controlled – warrior, elf or wizard. There are also a number of independent characters with whom these three can interact.

As in those from Beyond Software, the game takes place in real time so a fairly regular change of control from one to the other is advised, or you may find orcs





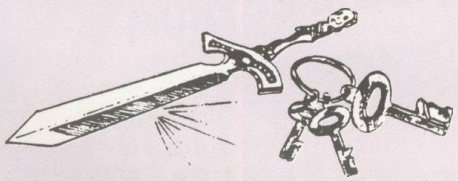


have overrun one of your comrades. Text input gives the feeling of being similar to The Hobbit, with what appears to be a large vocabulary and many complex inputs recognised and acted upon.

Elementary . . .

For those not so interested in swords, dragons and magic spells, there are a growing range of adventures that take place nearer to home. Melbourne House should satisfy those would-be detectives among us with Sherlock. The aim is to prove to that pillar of the law, Inspector Lestrade, that the solution to various crimes is elementary.

Lots of travel by hansom cab and steam train will get you and your friendly biographer to the scene of the



crime. Not an easy game to get into but persevere and you will find life in Baker Street quite exciting.

Another game of detection – but very much of today – is The Forth Protocol from Hutchinson Computer Publishing. This three part graphic adventure requires very few typed inputs. It uses a series of small pictures or icons that represent permitted commands and you simply move a pointer to the one you wish to choose.

The plot revolves around you having to track down possible espionage suspects and evaluate information obtained from a central security computer. Everything takes place in real time with the threatened detonation of a nuclear device as the incentive to solve the many clues on time. The innovative presentation combined with a slick plot makes this one for any serious collection.

Espionage is also the link with Mindshadow from Activision. Here the problem is to find out who you are. You were obviously involved with some form of organisation – but what? You wake up on a deserted isle having been attacked and lost your memory.

What happens next is up to you. There are good graphics, good responses to your commands and some excellent features that many software houses would do well to emulate. Spectrum users can only hope for more from the same source.

Far out

Moving further into the future there are several science fiction games that will keep you checking your oxygen and watching out for aggressive robots. Level 9 seems to have the monopoly on good adventures of this type and the Silicon Dream trilogy of Snowball, Return to Eden and The Worm in Paradise cannot be bettered at present.

The last two have good graphics and an operating system that, in the case of the Worm, is probably the best produced in the UK. The descriptive text is imaginative and very readable with an even greater dose of that very special Level 9 humour. Although not easy games to solve, they encourage players of any standard to continue and Level 9 offer the best cheat sheets in the business.

If humour is what you want then the satire of Hampstead or Terrormolinos from Melbourne House or Bored of the Rings from Silversoft will probably tickle your fancy. All are good games that have brought a new meaning to adventure games. Gone are the days where a sharp sword was the only requirement for a true adventurer.

As you can see the choice is vast, and here we have only scraped the surface to give you a selection of our favourites. Not all of them will appeal to everybody and the answer is to make up your mind which kind of adventure will suit you personally.

Save £2
-See Centre Pages

YOU can go for gold ...with the MICRO OLYMPICS

This is the package that broke all records! More than a game – it's a brilliantly written collection of ELEVEN great track and field events!

Ever imagined yourself as another Seb Coe? Then try to run against the world record holder at 1500 metres. And if that distance is too much for you then there's always the 100, 200, 400 and 800 metres to have a go at.

Not much good at running? Don't worry, MICRO OLYMPICS has many more challenges for you. Why not try your skill at the high jump or the long jump?

And if you can't beat the computer at running or jumping then you can always throw things around in frustration! The trouble is that it's just as hard to be a champion at the discus, the hammer or the javelin.

And the pole vault takes the event to new heights!

Yes, it's fast, furious fun, pitting yourself against the world's best times and distances on your micro.

You may not be another Steve Ovett or Alan Wells, but with practice you COULD become the Micro Olympics Champion!



Play Micro Olympics – and let your fingers do the running!

Send for it today

Please send me _____ copy/copies of Micro Olympics

☐ I enclose cheque made payable to Database Publications Ltd. for £ _____

Spectrum 48k £5.95

I wish to pay by
☐ Access ☐ Visa No. _____ Expiry date _____

Signed _____

Name _____

Address _____

_____

Post to: Micro Olympics offer, Database Publications, 68 Chester Road, Hazel Grove, Stockport SK7 5NY.

Now YOU can fly with the legendary Red Arrows – in the most challenging flight simulation ever!

It's the most exciting flight simulator ever written for a home computer – the product of many months of dedicated work by some of Britain's top programmers, enthusiastically aided by the talents of aircraft designers, engineers, mathematicians – and the Red Arrow pilots themselves.

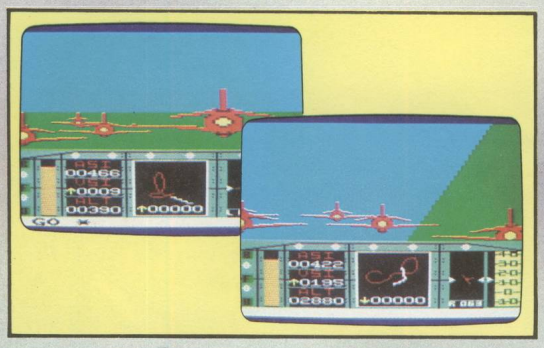
Every ounce of power contained in the micro, and its enhanced sound and graphics capabilities, is used to give the utmost realism to re-creating the most spectacular aeronautical displays ever seen in the skies of Britain.

You start by practising take offs and landings. Then, once you have won your wings, you fly in formation as part of the Red Arrows team. There's no margin for error as you fly a mere six to 10 feet from each other – at speeds of between 300 and 350 miles an hour!

But the real drama begins as you plunge into the death-defying manoeuvres that have been thrilling crowds at air shows for the last 21 years.

On the panel in front of you are all the instruments you need – plus a screen giving you an external view of the complete formation you are flying. Slip out of line for a second and the eagle-eyed Red Leader will be on the radio ordering you back into position.

The program comes with a detailed flight handbook that will soon give you the confidence to take YOUR place alongside the ace pilots of the Red Arrows, even if you've never flown before!



Save £2!

Now only £6.95

ORDER FORM

Spectrum 48k Spectrum+, Spectrum 128
Tape £6.95 *9005* ☐

I wish to pay by:

☐ Access/Mastercard/Eurocard No.

☐ Barclaycard/Visa No.

☐ Cheque/PO made payable to Database Publications Ltd.

Expiry date: /

Name _____

Address _____

_____

Signed _____

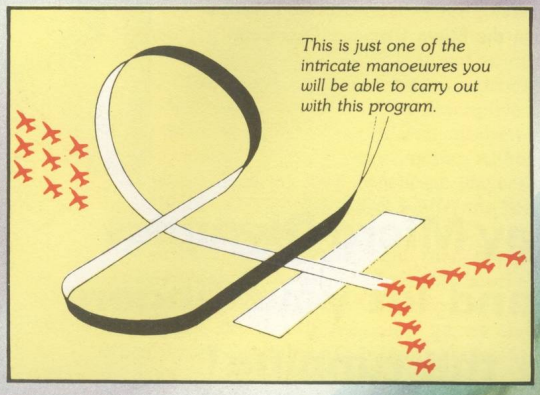
Order at any time of the day or night

Telephone Orders: **061-429 7931**

Orders by Prestel: **Key *89, then 614568383**

Don't forget to give your name, address and credit card number

ENQUIRIES ONLY: **061-480 0171 9am-5pm**



This is just one of the intricate manoeuvres you will be able to carry out with this program.

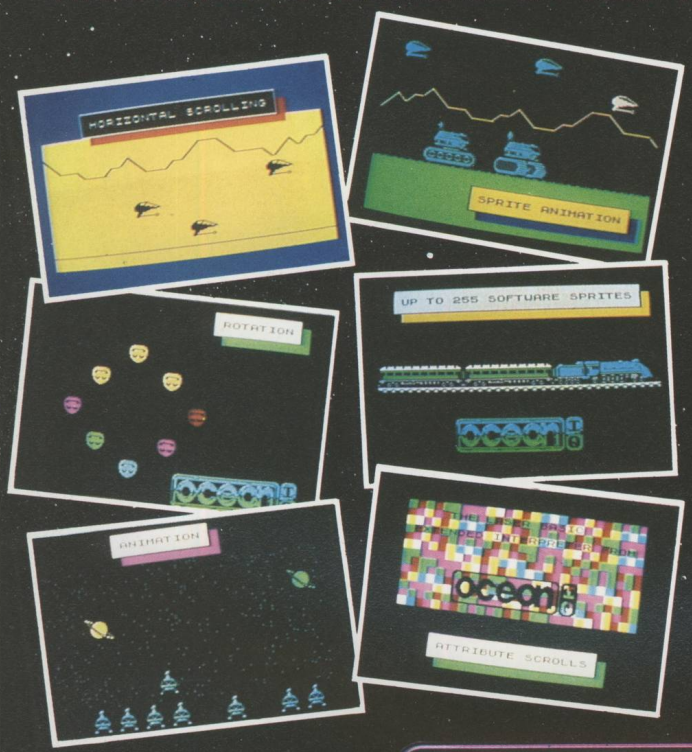
Put yourself in the pilot's seat of the most manoeuvrable fighter in the RAF!


RED ARROWS
"If you want to have some fun and say that you were flying in formation with the 'Big Nine' then this is the program for you."
– Amstrad User

Laser Basic

TURN YOUR COMPUTER INTO A PROFESSIONAL GAMES WRITING MACHINE FOR FUN AND PROFIT!

LASER BASIC adds 100 new commands to Sinclair Basic. These extended commands are semi-compiling so graphic animation is extremely fast. LASER BASIC includes, extended interpreter, sprite/graphic designer, fully documented program and 2 sets of pre-defined sprites.



- Up to 255 software sprites can be defined, each with its own user selectable dimensions (up to 7 screens wide!)
- Operations can be carried out, on or between screen windows, sprites and sprite windows.
- Sprites can be block 'PUT' or can use one of three logical operations – AND, OR and XOR.
- Sprites and screen windows can be pixel scrolled in any direction, mirrored, enlarged, spun, inverted or cleared.
- Procedures with local variables and parameter passing.
- TRON and TROF (trace facility).
- 16 bit PEEK and POKE.
- RENUMBER and REM renumber.
- Non destructive MOVE with 2 frame animation.
- Collision detection and pattern recognition facilities.

OUT NOW FOR THE SPECTRUM 48K/ SPECTRUM+ ON CASSETTE

£14.95

Microdrive compatible.

ALSO AVAILABLE NOW!



Spectrum 48K/ Spectrum+ **£9.95** on cassette. Microdrive compatible.

A new age dawns! The arrival of LASER BASIC: first in a powerful range of development tools for fast programming, brought to you with the combined skill and resources of Ocean and Oasis – (producers of "White Lightning".)
Other, easy to use products in this expanding range will include screen artist/designers, music composers and machine code emulators to make your programming more rewarding and exciting.

Ocean iQ INTERACTIVE SOFTWARE

Ocean iQ Software is available from selected branches of: WHSMITH, John Menzies, WOOLWORTH, LASKYS, Rumbelows, Greens. Spectrum Shops and all good software dealers. Trade enquiries welcome.

6 Central Street, Manchester M2 5NS. Telephone: 061-832 6633. Telex: 669977 Oceans G.

Specifically Spectrum

The last part of our exploration of the Z80 instruction set

PROGRAMMING the Z80 microprocessor is one thing, programming a particular computer which incorporates it is another. The machine code instruction set is always the same wherever a Z80 is used, but the things you can do with it are limited and dictated by the facilities offered by the rest of the computer.

Micros used in business usually employ a standard operating system which is common to a lot of computers of different makes. CP/M was the first of these transportable operating systems and the one which is most often used with Z80 (and the earlier 8080) based computers. More recently a number of new operating systems, like MSDOS and PCDOS, have sprung up for use with the various 16-bit microprocessors, but these have no relevance to the Z80 processor.

There are a number of features of the Spectrum hardware that make it impossible to use CP/M with the ordinary 48k Spectrum. This is not a terrible drawback because CP/M is not very well suited to the kind of thing that the Spectrum is most used for. CP/M is mostly concerned with handling text and disc files, with almost no graphics capability at all. This is fine for business use, but hopeless for video games.

When learning to program the Spectrum we have to learn more than just the Z80 machine language. We have to know a lot about the rest of the computer as well. Much of this knowledge will not be of use in programming other computers, which will have their own foibles and limitations. This is the real drawback of not having a standard operating system. What you learn may not be transportable to other machines.

It is a good idea to keep in mind the fact that almost everything about the Spectrum is non-standard. Clever hardware shortcuts and software routines have been used instead of handfuls of expensive peripheral chips. When you move from the Spectrum to a more expensive computer you will probably find programming a much simpler and more straightforward affair. It will certainly be different.

In and out

The aspects of the Spectrum in which most individuality is evident are the parts of the machine concerned with input and output. This does not mean just the reading of joystick ports and so on with the Basic IN and OUT commands, but all input and output – including cassette loading and saving, typing on the keyboard, and the display on the screen and ZX printer.

Each of these functions is handled in a highly idiosyncratic way on the Spectrum, and indeed other home computers also use unusual methods to do some of these things.

Luckily most of the hard work of writing code to handle the input and output has already been done, and the routines required are available in the Spectrum ROM. All we have to do is learn how to use them.

The screen

Whenever the Spectrum is turned on, the ULA is busy scanning the screen memory and converting the contents to a television picture. Memory elements set to a binary 1 cause a pixel to appear in the INK colour for that part of the screen, and elements set to zero cause a pixel to be displayed in the local PAPER colour. All you have to do to make things appear on the screen, therefore, is load the appropriate memory locations with the right patterns of 1s and 0s, and maybe also load the attribute memory locations with the desired colour codes.

A lot of graphics routines do just that – poke bytes into the screen memory. However the Spectrum screen memory is laid out in a funny, non-continuous fashion which isn't easy to work with, and a lot of effort can be saved with a ROM call, provided what you wish to put on the screen is text or a user-defined graphic.

Just load the A register with the code for the character you wish to print on the screen and then call the routine at &0010. There is a bit more to it than that, because you ought first to make sure that the right print channel is open, but this is easily done by calling another ROM routine.

Besides codes for characters and UDGs, you can "print" control characters to alter the printing position

```
32000  62    7D00  3E  3E  LD A,&02      ;Channel 2 is the
32001  2     7D01  02  02                ;main screen
32002  205   7D02  CD  CD  CALL &1601     ;CHAN-OPEN ROM
                                          routine
32003  1     7D03  01  01
32004  22    7D04  16  16
32005  62    7D05  3E  3E  LD A,"A"       ;Code for "A" is &41
32006  65    7D06  41  41
32007  215   7D07  D7  D7  RST &0010      ;Call print routine
32008  62    7D08  3E  3E  LD A,"B"       ;Code for "B" in A
32009  66    7D09  42  42
32010  215   7D0A  D7  D7  RST &0010      ;Print it
32011  62    7D0B  3E  3E  LD A,"C"       ;Now a "C"
32012  67    7D0C  43  43
32013  215   7D0D  D7  D7  RST &0010
32014  201   7D0E  C9  C9  RET           ;Return
32015  0     7D0F  00  00
```

Program I

or printing colours. Try out the bit of code in Program I with the hex handler. Note that RST 0010 is one of the special one byte call instructions, and is the equivalent of CALL &0010.

If you want to print a number of characters at once, it may be easier to use the PRINT-STRING routine. This time we load two pointers before calling the ROM routine. DE holds the address of the start of the string, and BC holds the number of characters in the string.

Notice that the space at the end of the string means

```
32100  62   7D64  3E  3E  LD A,&02      ;Select channel 2
32101  2    7D65  02  02
32102  205  7D66  CD  CD  CALL &1601    ;Open Channel
32103  1    7D67  01  01
32104  22   7D68  16  16
32105  17   7D69  11  11  LD DE,&7D75   ;Start of text
32106  117  7D6A  75  75
32107  125  7D6B  7D  7D
32108  1    7D6C  01  01  LD BC,&0006   ;Length of text
32109  6    7D6D  06  06
32110  0    7D6E  00  00
32111  205  7D6F  CD  CD  CALL &203C    ;Call PRINT-STRING
32112  60   7D70  3C  3C
32113  32   7D71  20  20
32114  201  7D72  C9  C9  RET           ;Return
32115  0    7D73  00  00                ;Waste of
32116  0    7D74  00  00                ;space
32117  72   7D75  48  48  DEFB "H"      ;Text stored here
32118  101  7D76  65  65  DEFB "e"
32119  108  7D77  6C  6C  DEFB "l"
32120  108  7D78  6C  6C  DEFB "l"
32121  111  7D79  6F  6F  DEFB "o"
32122  32   7D7A  20  20  DEFB " "     ;Space
32123  0    7D7B  00  00
```

Program II

that successive calls to the routine in Program II will print the word Hello several times across the screen separated by spaces. If you change the space to a carriage return (&0D), successive Hellos will be printed on a new line.

There are also a couple of handy ROM routines for printing numbers. This is not as easy to do as you might think at first, as it involves changing the contents of a binary register into a string of character codes representing the decimal equivalent, and then printing that. Using the ROM takes all the sweat out of the process.

Program III demonstrates the use of the ROM routine that prints the Basic line numbers. It is only good for numbers up to 9999, but does have the

```
32200  62   7DC8  3E  3E  LD A,&02      ;Again, open channel
32201  2    7DC9  02  02                ;2
32202  205  7DCA  CD  CD  CALL &1601
32203  1    7DCB  01  01
32204  22   7DCC  16  16
32205  1    7DCD  01  01  LD BC, &04D2  ;Decimal 1234
32206  210  7DCE  D2  D2
32207  4    7DCF  04  04
32208  205  7DD0  CD  CD  CALL &1A1B    ;Call OUT-NUM
32209  27   7DD1  1B  1B
32210  26   7DD2  1A  1A
32211  201  7DD3  C9  C9  RET           ;That's all
```

Program III

advantage that it prints leading spaces for numbers with fewer than four decimal digits. The number to be printed must be in BC when the routine is called.

More complex numbers, including full floating point notation, can be printed with the ROM PRINT-FP routine. Using floating point arithmetic on the Spectrum involves calling the ROM calculator routines, which is a very big subject and beyond this article. PRINT-FP simply prints the top number on the calculator's own stack.

Unless you are heavily into mathematics, the biggest number you will probably ever want to print is the maximum contents of a double register, &FFFF or 65535 decimal. What you do is load the number into BC, and call the STACK-BC routine to put BC onto the calculator stack in floating point notation, then PRINT-FP will print it out. Program IV shows how.

Note that we have been using channel two throughout to direct the printout to the main screen. You can use other channels as well, for instance channel one for the bottom two lines of the screen, or

```
32300  62   7E2C  3E  3E  LD A,&02      ;As Usual
32301  2    7E2D  02  02
32302  205  7E2E  CD  CD  CALL &1601
32303  1    7E2F  01  01
32304  22   7E30  16  16
32305  1    7E31  01  01  LD BC,&FFFF
32306  255  7E32  FF  FF
32307  255  7E33  FF  FF
32308  205  7E34  CD  CD  CALL &2D2B    ;Call STACK-BC
32309  43   7E35  2B  2B
32310  45   7E36  2D  2D
32311  205  7E37  CD  CD  CALL &2DE3    ;Call PRINT-FP
32312  227  7E38  E3  E3
32313  45   7E39  2D  2D
32314  201  7E3A  C9  C9  RET           ;All done
```

Program IV

channel three for the ZX printer, or even the extended channels with Interface 1.

The keyboard

Reading the keyboard is a complex procedure in the Spectrum, involving the reading of eight separate input ports and considerable bit testing and calculation to take account of the use of shift keys and so on. However the Spectrum normally takes care of this process automatically every 50th of a second or so, as long as the interrupts are left enabled. Many machine code programmers find it convenient to allow this automatic keyboard scanning to continue, and just pick up the code of the last key pressed from the systems variable at location 23560.

There are a few disadvantages with this approach. Advanced programming techniques sometimes use the interrupt system for other purposes, like continuous music. It may be desired to save processing time by disabling the interrupts, especially when only a small portion of the keyboard needs to be read, as in most games. There is also the fact that the "last key" code in location 23560 can only register one keypress at a time, like the INKEY$ function in Basic, whereas things like diagonal joystick movement require the simultaneous reading of two keys.

Reading the keyboard directly for your own programs need not be as involved as it is when the Spectrum does it. Normally you will only be interested in a few of the keys at any given time, so you won't have to scan the whole keyboard, and you are unlikely to be concerned with things like the extended mode shifts. The eight keyboard ports and their respective keys are laid out in Figure I.

If the key is pressed, the appropriate bit will be set to zero, if it is not pressed, the bit will be a one. Bits 5 and 7 are not used, but must not be assumed to be either one or zero, as they can be either value at random on some Spectrums. Bit 6 is read from the EAR socket, and also may be either one or zero, depending on what is going on there.

In Program V the port connected to keys 1 to 5 is

Port Number	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
32766-&7FFE	B	N	M	SYM SH	SPACE
49150-&BFFE	H	J	K	L	ENTER
57342-&DFFE	Y	U	I	O	P
61438-&EFFE	6	7	8	9	0
63486-&F7FE	5	4	3	2	1
64510-&FBFE	T	R	E	W	Q
65022-&FDFE	G	F	D	S	A
65278-&FEFE	V	C	X	Z	CAPS SH

Figure I

read continually until one of these keys is pressed, whereupon the number read (which will have been complemented and masked) is printed on the screen. In a real program, you would probably test for certain keypresses and use the results to make the program execution jump to an appropriate routine.

```
32400  62   7E90  3E  3E  LD A,&02      ;Open channel 2
32401  2    7E91  02  02
32402  205  7E92  CD  CD  CALL &1601
32403  1    7E93  01  01
32404  22   7E94  16  16
32405  1    7E95  01  01  LD BC,&F7FE   ;Port number 63486
32406  254  7E96  FE  FE
32407  247  7E97  F7  F7
32408  237  7E98  ED  ED  IN A,C        ;Read the port
32409  120  7E99  78  78
32410  47   7E9A  2F  2F  CPL           ;Complement A
                                        register
32411  230  7E9B  E6  E6  AND &1F       ;Mask off extra bits
32412  31   7E9C  1F  1F
32413  40   7E9D  28  28  JR Z,&F6      ;Jump back to 32405
32414  246  7E9E  F6  F6                ;if no keypress
32415  6    7E9F  06  06  LD B,&00      ;Clear B register
32416  0    7EA0  00  00
32417  79   7EA1  4F  4F  LD C,A        ;A goes into C
32418  205  7EA2  CD  CD  CALL &1A1B    ;Call PRINT-NUM
32419  27   7EA3  1B  1B
32420  26   7EA4  1A  1A
32421  62   7EA5  3E  3E  LD A,&0D      ;Carriage return
32422  13   7EA6  0D  0D
32423  215  7EA7  D7  D7  RST &0010     ;Print it
32424  201  7EA8  C9  C9  RET           ;Back to Basic
```

Program V

Sound

Writing to port 254 drives both the BEEP speaker (bit 4) and the MIC socket (bit 3), and can also be used to change the border colour (bits 0, 1, and 2). All three are affected at once by the same OUT instruction, so if you want to make a sound without changing the border colour, you have to be sure to set the border colour bits to the existing colour.

If you just want to make musical notes, it is probably easiest to load the appropriate values into registers and call the beeper routine in the ROM, but to get weirder sound effects and noises it is better to turn the speaker and MIC bits on and off directly, and use your own timing loops to get sliding tones and the like. This is a great area for experimentation, and some programmers have even managed to get limited voice synthesis from a standard Spectrum by directly manipulating this port.

Sound routines should be located in the higher address memory beyond location 32768. The stack must also be located beyond this address, otherwise there will be a ragged quality to the sound. This is because the ULA has priority over the CPU at RAM addresses below 32768, a consequence of the video circuitry. The interrupts should also be disabled to avoid a 50-cycle buzz being overlaid on the desired sound.

When using the ROM beeper routine, however, both of these problems are handled by the ROM and you needn't worry about them.

To use the beeper in Program VI, DE is loaded with a counter to set the duration, and HL is loaded with a number that governs the pitch of the note. Both of these numbers are calculated from the frequency of the desired note. DE is set to a number equal to the desired duration (in seconds) times the frequency. HL is set to a number equal to 437,500 divided by the frequency, less 30. So in our example, half a second of the note A above middle C – sometimes called "International A" – with a frequency of 440 Hz requires 1/2 times 440 in DE, and (437,500/440)–30 in HL. Yes, I know it's a bind – that's what makes it such fun!

```
32500  17   7EF4  11  11  LD DE,&00DC   ;Duration timer
32501  220  7EF5  DC  DC
32502  0    7EF6  00  00
32503  33   7EF7  21  21  LD HL,&03C4   ;Pitch value
32504  196  7EF8  C4  C4
32505  3    7EF9  03  03
32506  205  7EFA  CD  CD  CALL &03B5    ;Call the beeper
32507  181  7EFB  B5  B5
32508  3    7EFC  03  03
32509  201  7EFD  C9  C9  RET           ;That's all, folks
```

Program VI

The next example, Program VII, is a simple rising note sound effect, with border colour changes incorporated. This time we ignore the ROM routines and drive the port direct. Zap!

```
33000  243  80E8  F3  F3  DI            ;Disable interrupts
33001  14   80E9  0E  0E  LD C,&FF       ;Counter
33002  255  80EA  FF  FF
33003  62   80EB  3E  3E  LD A,&1A       ;MIC on, Border red
33004  26   80EC  1A  1A
33005  211  80ED  D3  D3  OUT (&FE),A    ;Send to Port &FE
33006  254  80EE  FE  FE
33007  65   80EF  41  41  LD B,C         ;Counter to B
33008  16   80F0  10  10  DJNZ &FE       ;Countdown and loop
33009  254  80F1  FE  FE                 ;to 33008
33010  62   80F2  3E  3E  LD A,&04       ;MIC off, Border
                                         green
33011  4    80F3  04  04
33012  211  80F4  D3  D3  OUT (&FE),A    ;Send it
33013  254  80F5  FE  FE
33014  65   80F6  41  41  LD B,C         ;Counter to B again
33015  16   80F7  10  10  DJNZ &FE       ;Loop to 33015
33016  254  80F8  FE  FE
33017  13   80F9  0D  0D  DEC C          ;reduce counter
33018  32   80FA  20  20  JR NZ,&EE      ;Jump back to 33003
33019  238  80FB  EE  EE                 ;if counter not zero
33020  251  80FC  FB  FB  EI            ;Enable interrupts
33021  201  80FD  C9  C9  RET           ;To Basic
```

Program VII

And so on

The Spectrum ROM contains almost 16k of code routines – far more than we can cover in this series. If you wish to get further into Spectrum machine code, you will need to obtain a few reference books on the subject.

The most important of these is the assembly listing of the Spectrum ROM itself. This is available from Melbourne House under the title "The Complete Spectrum ROM Disassembly". It is by no means easy reading, but many people believe that this book is one of the major reasons why there is so much good software for the Spectrum.

There are other books about machine code and the Spectrum, all with something to contribute, but another source of information is other people's programs. These are not always easy to investigate, as most software has some kind of protection nowadays, but some of the earlier game programs from a few years ago are fairly easy to get into and the effort is well worthwhile.

Our faithful hex handler can be used to examine other people's code, but a disassembler would be preferable, as it will render the opcodes into more readable assembler mnemonics.

Don't be shy about breaking in and investigating protected software. This is neither illegal not immoral – as long as you don't make copies – and everyone does it, including the top programmers!

Smileys all round

The finishing touches to our well-structured game

LAST time we met our friends called Smileys. We started to put together a simple game and the idea was to structure the program to demonstrate the benefits of such programming. This should have enabled you to follow it through line by line and – provided you have made no typing erros – when you RUN it you should see 10 asterisks displayed in the main grid showing the locations of our temporarily-revealed hidden Smileys. These will soon be concealed again, but just for the time being we'll leave them showing so that we can test the program works correctly.

Right then, we've drawn the grid, initialised the array elements and hidden the Smileys in 10 random locations. We are now ready for the input routine.

What goes in must come out

First of all remove our temporary line 75 and type in lines 1010 to 1080.

```
1010 PRINT AT 3,0; INK 1;"Input x,y -
    ";
1015 IF INKEY$<>"" THEN  GO TO 1015
1020 LET k$=INKEY$: IF k$<"0" OR k$>"
9" THEN  GO TO 1020
1030 LET guessx=VAL (k$): PRINT AT 3,
20; INK 1;guessx;",";
1040 IF INKEY$<>"" THEN  GO TO 1040
1050 LET k$=INKEY$: IF k$<"0" OR k$>"
9" THEN  GO TO 1050
1060 LET guessy=VAL (k$): PRINT AT 3,
22; INK 1;guessy
1070 LET turns=turns+1
1080 PRINT AT 12,3;turns
```



These deal solely with our input to the computer. Line 1010 prints the text prompt where originally the title was displayed. It simply asks us to input the coordinates of our guess, column first, then row. Lines 1020 and 1050 use the INKEY$ command to await our entry, and lines 1015 and 1040 ensure that INKEY$ is empty or null before we actually press a key.

It's worth noting here, by the way, that lines 1020 and 1050 won't let us go any further until we press a key with a value between 0 and 9 inclusive. This is called validating the input and is another important factor that sorts out good programming from bad. There are people who delight in being able to crash such an input with entries like –255, 99999999.99 or even King Kong. A good programmer will put the necessary checks in an input routine to make sure that nothing, other than the response he wants, gets through.

Using the VAL command we assign the numeric value of the input string to the variables *guessx* and *guessy* then print these out with a comma already between them. Lines 1030 and 1060 are responsible for this. Line 1070 simply adds 1 to the variable *turns* to keep tabs on how many attempts we've had, and we use this to update the numbers of guesses on the screen display.

Unfortunately you can't really check whether this routine is working correctly at this stage as it needs a routine to process the information you have input.

Is there anybody there?

So let's continue by typing in the check routine from lines 1210 to 1330. As this calls another subroutine starting at line 1700 you must also type in lines 1710 and 1720 as well.

```
1210 IF INKEY$<>"" THEN  GO TO 1210
1220 IF b(guessx+1,guessy+1)=2 THEN
BEEP .2,4: BEEP .5,0: PRINT AT 21,6;
INK 0;"You've had that one !": FOR x=
1 TO 100: NEXT x: PRINT AT 21,6;"
                  ": RETURN : REM 21 s
paces
1230 LET flagx=0: LET flagy=0
1240 IF b(guessx+1,guessy+1)=1 THEN
GO SUB 1700: GO TO 1260
1250 PRINT AT guessy+10,guessx+9;" ":
 FOR z=10 TO 0 STEP -1: BEEP .01,z: N
EXT z: LET b(guessx+1,guessy+1)=2
1260 FOR x=1 TO 10: IF b(x,guessy+1)=
1 THEN  LET flagy=1
1270  NEXT x
1280 FOR y=1 TO 10: IF b(guessx+1,y)=
1 THEN  LET flagx=1
1290 NEXT y
1300 IF flagx=1 AND flagy=0 THEN  PRI
NT AT 21,9;"Right column"
1310 IF flagy=1 AND flagx=0 THEN  PRI
NT AT 21,9;"Right row"
1320 IF flagy=1 AND flagx=1 THEN  PRI
NT AT 21,6;"Right column & row"
1330 FOR x=1 TO 100: NEXT x: PRINT AT
 21,6;"                  ": REM 18 sp
aces
```

```
1710 PRINT AT guessy+10,guessx+9; INK
 0;CHR$ 145: FOR z=20 TO 40: BEEP .01
,z: NEXT z: LET smileys=smileys+1: LE
T b(guessx+1,guessy+1)=2
1720 PRINT AT 15,27;smileys
```

We now have to check the input to see whether or not we have found a Smiley. To do this the micro checks our selected element of the array *b()* for certain numbers, and responds according to what it finds there.

You may remember that we originally put 0s in all the array locations, then scattered some Smileys randomly around the grid in the form of 1s. In order to provide more information I've also introduced the number 2 to indicate that a particular location has already been chosen. Line 1220 first of all checks the

x,y location in the array for that number 2. If if finds one we sound a couple of beeps then print a message to the effect that we have previously chosen that location. After a short pause the message is blanked out and the RETURN takes us back immediately for another input.

If a 2 is not encountered, line 1230 sets up two flags *flagx* and *flagy*. These are used later to provide some useful clues if and when you select an element containing a 0, or in other words a blank.

Line 1240 continues the search and if a 1 is found – one of our hidden Smileys – the subroutine at line 1700 is immediately called. This prints our smiling face character, CHR$(145), at the location selected, accompanied by a suitably triumphant noise.

Then the variable *smileys* is incremented by 1 to be printed out on the screen by line 1720. Finally a 2 is put in the array element to indicate that it has been used.

We then RETURN to the original subroutine to go through the clue sequence. This is designed to provide information on the location of any Smiley in the same row or column as the current guess.

If a location is found with a 0 in it, line 1250 prints a space in the grid. We put a 2 in the array element and the program goes straight through to the clue sequence.

This uses two FOR ... NEXT loops to scan the rows and columns, and sets *flagx* or *flagy* depending on what it finds. For instance, if there is a Smiley on the same row that you selected, *flagx* is set to 1. If there's one in the same column, *flagy* is set, and naturally, if there's one in both column and row, both flags are set.

Lines 1300 to 1320 then print out one of three simple messages corresponding to the contents of these flags which is removed by line 1330 after a short delay.

Testing, testing

Now comes the bit you've been waiting for – a quick check that the game is running correctly so far. Enter the following lines:

```
130 IF smileys<3 THEN GO TO 100
135 STOP
```

Now you can test your input and detection routines by running the program again. You can try out any figure you want in place of the 3 in line 130, but I would suggest that to save time you only use a small number to start with.

When that number is reached on your Smiley count, the program will drop through to the STOP statement at line 135, because the game continues condition, *smileys<3*, no longer exists. You can probably see now the reason for indicating the whereabouts of the Smileys – it does make testing so much easier. But don't just bash away getting bullseyes all the time. It's imperative that you also test the miss routine and occasionally select a location that has already been used.

If after one or two tests you are happy that everything is working smoothly, remove line 135 and change the number 3 in line 130 to 10, because that's the number of Smileys we've got to find before the game finishes.

That's all folks

We are now ready for the routine called by line 140 to signify the end of the game. Type in lines 1410 to 1470 and lines 150 and 160.

```
1410 BEEP .5,1: PRINT AT 21,7; INK 0;
"That's the lot  !!"
1420 PAUSE 100
1430 IF turns<best THEN  LET best=tur
ns
1440 PRINT AT 18,3;best
1450 FOR x=1 TO 200: NEXT x: PRINT AT
 21,7;"Another game y/n ?"
1460 LET k$=INKEY$: IF k$<>"Y" AND k$
<>"y" AND k$<>"N" AND k$<>"n" THEN  G
O TO 1460
1470 LET again=0: IF k$="Y" OR k$="y"
 THEN  LET again=1
```

```
150 IF again=1 THEN  GO TO 60
160 STOP
```

Line 1410 immediately prints a message indicating that we've found the last Smiley and your score is then checked by line 1430 to see whether it qualifies as the best so far. Line 1450 prompts for another game and uses the INKEY$ command again to await your response validated for presses of only the Y and N keys. If the response is affirmative the variable *again* is set to 1.

Returning from the subroutine takes us to line 150, which, should it find *again* set, sends the program to line 60 to re-run the game completing our main loop. This carefully avoids the variable *turns* at line 20 which must not be reset otherwise a best score display would be pointless.

It also avoids the array at line 30 which cannot be redimensioned unless it has been cleared. There is nothing to be achieved in doing this in these circumstances so it might as well be left alone. We also avoid the routine that would redefine the two graphic characters as this is time wasting and unnecessary.

Should line 150 find the variable *again* equal to 0 – a negative response – line 160 throws you unceremoniously out of the program, and quite rightly too.

It's not every day you get the chance to play something as exciting and infinitely rewarding as Smiley Hunt.



Now you see me, now you don't

If, having played the game a few times, you are happy that it is working correctly you can remove line 845, the one that shows you where the Smileys are. You'll find it's a different game altogether now! It may not be the most sparkling program in the world, but it works, and contains some useful techniques. For instance, the input routine could be used in any program – it probably already has been – and the checking routines could always prove useful.

The important thing to us, though, is that it is well structured and as a result you should have been able to follow it through line by line, subroutine by subroutine, and see how it was put together.

I hope you enjoy playing it, but even more I hope you've learned something about structured programming style by typing it in. There's a popular song that says: "You've either got or you haven't got style!" Believe me, if your programs have it, progressing on to something more ambitious will not be the enormous step you may have imagined.

On the move...

Final part of the series which clearly explains the complex area of machine code graphics

SO far in this series we've looked at the display file, the section of RAM responsible for the characters' shapes, and the attribute file which is responsible for their colour. Now we'll see how to get things moving.

There are two listings this month, though they're actually the same program. Program I is an Assembly language listing of the demonstration and Program II is the same program with the machine code stored in data statements.

If you've got an assembler enter Program I. If not type in and run Program II.

When run you'll see some text printed down the centre of the screen and a large character on the left. Hold down the 2 key and the character will move smoothly across the screen to the right. Hold down 1 and it moves back again.

There are a couple of points worth noting. As you've discovered if you've just tried the program, moving things around in machine code isn't like Basic. When I first wrote the program the character whizzed off the screen so fast I never saw it move – I must have blinked at the time. You'll see from Program I that a 0 to 2000 delay loop is executed every time the character moves to slow it down, otherwise it's uncontrollable.

There are no boundaries round the screen and if the character slides off one side it reappears on the other This is because we aren't using coordinates like Basic, just an address in RAM at which the character data is to be stored.

One way of keeping it on the screen is to store its *x* coordinate yourself along with the address. Every time the character moves to the right increment *x* so you can check *x* before the character is moved and if it's at the right edge you know it can't be moved any further. Similarly, you can just as easily check for the left edge.

Another point worth noting is that the character moves smoothly through the text in the centre of the screen without affecting it. That's because an XOR print routine is used.

The second instruction after *loop5* in *print* is XOR (HL). This XORs the character data with the screen data. You can remove this if you want and do a straight poke. The character will then destroy any background it passes over. The advantage of a straight poke is that it's faster.

Call *print* with the print address in HL, the character data address in DE and the size in BC. The number of columns is in C and the rows in B. Calling it once places the character on the screen. Calling it again with the same parameters erases it.

For a further explanation of the print routine take a look at the first article in this series – it's the same one.

There's only one other routine used and you haven't met this before – it's called *key*. Call this when you want to read the keyboard. If a key is being pressed it returns with the Ascii code or token in the A register and the Carry flag clear. If carry is set then either there aren't any keys being pressed or too many are being held down. Either way the result in A is garbage.

Have a look at *key*. Three calls to the ROM are required plus some fiddling in between to find the Ascii code or token for the key being pressed.

The first call to &28E scans the keyboard and returns an intermediate code in the D and E registers. If the code is a valid result the Zero flag is set, otherwise it's clear. You'll see that straight after the call in key there's a JR NZ to *nokey* to check for this. The Carry is set indicating an invalid result and the routine ends.

Next there's a call to &31E. This converts the intermediate code into an offset into a look-up table. Obviously keys like Caps Shift do not produce proper codes so these are weeded out. This is indicated by the Carry being clear when the call returns. Again *key* checks for this and jumps to *nokey* if necessary.

A call to &333 is made to convert the offset into an Ascii code or token. The result is dependent upon the values of the registers on entry. Table I shows the entry conditions required. Finally, the Carry is cleared indicating a valid result and *key* ends.

There are other ways of reading the keyboard, but you should find this general routine satisfactory for most purposes.

E register = offset (from call to &31E)	
No Shifts	B=&FF
Caps Shift	B=&27
Symbol Shift	B=&18
G mode	C=&02
E mode	C=&01
K mode	C=&00:D=&00
L mode	C=&00:D=&08:Bit 3 of FLAGS 2, (&5C6A)=0
C mode	C=&00:D=&08:Bit 3 of FLAGS 2, (&5C6A)=1

Table I: Entry conditions for the ROM routine at &333

The main loop checks for Ascii 1 and Ascii 2, and either moves the character left or right. You'll see that there's also an additional check for Ascii 3 which I've set up as an Escape key to return to Basic.

No doubt you've already discovered when using commercial software written in machine code that pressing the Break key has no effect at all. It's Basic that checks this and since we're not using Basic it's not checked at all.

You have to do all the work yourself when in machine code. If you don't specifically test the Break key now and then, or any key for that matter, and take action when it's pressed then nothing will happen. It's a point worth bearing in mind when writing a machine code routine – always make sure there's a way out, otherwise the only solution is to pull the mains plug.

Obviously there's a lot more to writing a machine code arcade game than the routines presented here. There are score routines, high score tables, animation and so on. It's not so difficult once you know how the screen is mapped, have a print routine and know how to read the keyboard.

I've started you off, now it's up to you to carry on.

```
ORG 30000


LD HL,&4000     ;start address

.loop
LD (address),HL ;store
LD DE,data      ;data address
LD BC,&1002     ;rows/columns
CALL print      ;print new chr
LD HL,&800      ;0-&800 delay
.delay
DEC HL
LD A,H
OR L
JR NZ,delay
.wait           ;wait for a key
CALL key
JR C,wait
LD HL,(address)
CP "1"          ;check direction
JR Z,left
CP "2"
JR Z,right
CP "3"          ;end?
JR NZ,wait
RET

.left           ;go left
DEC HL
JR move
.right          ;go right
INC HL
.move
PUSH HL         ;save new
LD HL,(address) ;get old
LD BC,&1002     ;size
LD DE,data      ;data address
CALL print      ;erase old chr
POP HL
JP loop         ;again

;******** Keyboard Read *****
;Carry=1...no key pressed
;Carry=0...key pressed, A=chr

.key
CALL &28E       ;scan keyboard
JR NZ,nokey     ;valid result?
CALL &31E       ;get code
JR NC,nokey     ;valid?
LD E,A
LD C,0
LD D,8
CALL &333       ;get final code
AND A
RET
.nokey          ;key not pressed
SCF
RET
;************* Print ***************
.print

;HL=address
;DE=data
;BC=rows/columns

.loop4
PUSH BC         ;save counter
PUSH HL
.loop5
LD A,(DE)       ;get data
XOR (HL)        ;XOR with screen
LD (HL),A       ;store
INC HL
INC DE
DEC C           ;columns-1
JR NZ,loop5
POP HL          ;next row
LD A,H
AND 7
CP 7
JR Z,bottom
INC H
JP here
.bottom
LD A,H
RRCA
RRCA
RRCA
LD H,A
LD BC,&20
ADD HL,BC
SLA H
SLA H
SLA H
.here
POP BC
DJNZ loop4      ;rows-1
RET


.address DEFW 0


;***** Character Data *******
.data           ;16 x 2

DEFB 125,190,130,65
DEFB 130,65,154,89
DEFB 154,89,130,65
DEFB 124,62,128,1
DEFB 184,29,167,229
DEFB 144,9,143,241
DEFB 96,6,31,248
DEFB 24,24,120,30

END
```

Program I

```
 10 REM PROGRAM II
 100 CLEAR 29999
 105 INK 1: BRIGHT 0: PRINT "Thinking
..."
 110 LET p=30000
 120 LET a$=""
 130 IF a$="" THEN  READ a$: IF a$="S
TOP" THEN  GO TO 500
 140 LET y=CODE a$-48: IF y>9 THEN  L
ET y=y-7
 150 LET x=CODE a$(2)-48: IF x>9 THEN
  LET x=x-7
 160 POKE p,16*y+x
 170 LET p=p+1
 180 LET a$=a$(3 TO )
 190 GO TO 130
 200 DATA "2100482280751184750102100D
8775"
 205 DATA "2100082B7CB520FB"
 210 DATA "CD717538FB2AB075FE312809FE
322808FE3320ECC9"
 220 DATA "2B180123E52AB07501021011B4
75CD8775E1C33375"
 230 DATA "CD8E02200FCD1E03300A5F0E08
1608CD3303A7C937C9"
 240 DATA "C5E51AAE7723130D20F8E17CE6
07FE0728042AC3AC757C0F0F0F67012000090C
B24CB24CB24C110D8C9"
 250 DATA "00000000"
 260 DATA "7DBE824182419A599A5982417C
3E8001B81DA7E598098FF160061FF81818781
E"
 300 DATA "STOP"
 400 REM ------------------
 500 CLS : INK 2: BRIGHT 1
 510 PRINT AT 5,16;"S";AT 6,16;"P";AT
 7,16;"R";AT 8,16;"I";AT 9,16;"T";AT
10,16;"E";AT 11,16;"S";AT 12,16;"?"
 520 LET x=USR (30000)
 530 BRIGHT 0
```

Program II

Index

A
Adventures – creating 202
– playing 198
– software 206
Analysis 61, 144, 169
Animation 62, 84, 118
AND 194
Arrays 149, 182
Assembly language 105, 172
AT 22

B
Basic interpreter 160
Binary 30, 32, 58, 173, 194
Beginners 2, 38, 74, 110, 146, 182
Books 72
BORDER 2, 55
BRIGHT 22, 134
Bulletin boards 152

C
Cassette tape 5, 180
Character set 27
CHR$ 27
CIRCLE 96
Colour 54, 134
Communications 150
Conditional operators 183
CPU 16

D
DATA 146
Databases 90
DIM 149
Disc drives 42
DRAW 96

E
Editing 3, 8, 41
Education 186
Educational software 187, 189
Electronic mail 151, 156
EOR 194
Error handling 191
Etch-a-sketch 129
Extended mode 11

F
Fault finding 12, 107
Flags 170
FLASH 22, 134
FOR 110

G
Graphics 28, 57, 94, 122, 164, 124
Graphics hardware 114, 162, 178
Graphics software 130

H
Hacking 158
Hexadecimal 66, 98, 103

I
INK 22, 54
INPUT 76
Inside the Spectrum 24, 117
Interfaces 78, 126, 153
INVERSE 22, 135

J
Jargon 16
Joysticks 126

K
Keyboard upgrades 88

L
LOAD 5, 180
LET 38
LIST 22
Loops 76, 110, 183

M
Machine code 32, 66, 103, 122, 140, 164, 170, 209, 214
Memory map 122, 195
Memory upgrade 70
MERGE 180
Microdrive commands 49
Microdrives 45, 190
MicroLink 156
Micronet 150
Mnemonics 69, 104
Modems 153
Mouse packages 114

N
NEXT 110

O
OR 196
OVER 22, 135

P
PAPER 22, 54
PEEK 68
PLOT 95
Plotter 178
POKE 68
Prestel 150
PRINT 7, 22, 54, 134
Printers 78, 81

R
RAM 16, 192
READ 146, 182
Repairs 107
ROM 16, 209

S
SAVE 5, 180
Screen 94, 122, 209
SCREEN$ 23, 180
Sir Clive Sinclair 35
Spectrum 128 100, 117, 176
Speech 101
Spreadsheets 137
Stack 141
STEP 111
Strings 8, 74, 185
Structured programming 166, 212
Systems 18, 20

T
TAB 22

U
ULA 17
User defined graphics 28
USR 69

V
VERIFY 5, 180
Video monitors 52

X
XOR 197

Z
Z80 registers 25, 140
Z80 instruction set 24, 142, 170

Make sure you've got a COMPLETE set of

The COMPLETE SPECTRUM
All you want to know about the world's best-selling computer
A Database Publication

If you missed the first five issues you can order them for just £7.50 for all five. Single copies of the issues already published can be sent to you for £1.75 (including postage). Please use the order form below.

A dust cover for your *Spectrum and *Spectrum Plus

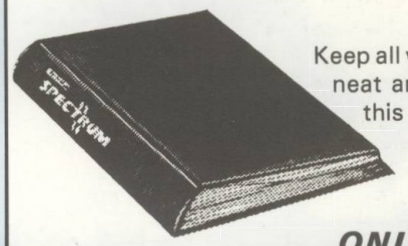


Made of soft pliable water-resistant black vinyl and decorated with *The Complete Spectrum* logo in silver.

*** Please ensure that you order the correct dust cover for your model.**

ONLY £3.95

A luxury binder for The Complete Spectrum



Keep all your copies neat and clean in this handsome binder.

ONLY £3.95

Valid to July 31, 1986

ORDER FORM

Please enter quantity as required in box

			£	p
The Complete Spectrum				
Issues 1-5 UK £7.50, Europe £11, Overseas £13	9111			
Single issues:				
No. 1	9016			
No. 2	9017			
No. 3	9018			
No. 4	9019			
No. 5	9020			
UK £1.75, Europe £2.25, Overseas £2.75				
Micro Olympics				
Tape £3.95	9003			
Mini Office				
Tape £3.95	9001			
Red Arrows				
Tape £6.95	9005			
Binder				
UK £3.95	9030			
Europe/Overseas £6.95	9031			
Overseas £10.95	9032			
Dust Covers				
*Spectrum Dust Cover (standard 16/48k)				
UK £3.95	9035			
Europe/Overseas £6.95	9036			
Spectrum Plus Dust Cover				
UK £3.95	9037			
Europe/Overseas £6.95	9038			
***Please ensure that you order the correct dust cover for your model**			TOTAL	

Send to: The Complete Spectrum
FREEPOST,
Europa House,
68 Chester Road,
Hazel Grove,
Stockport SK7 5NY.

Order at any time of the day or night

Telephone Orders:	*Orders by Prestel:*	*MicroLink/Telecom Gold*
061-429 7931	**Key *89, then 614568383**	**72:MAG001**

Don't forget to give your name, address and credit card number

ENQUIRIES ONLY: 061-480 0171 9am-5pm

Payment: Please indicate method (✓)

☐ Access/Mastercharge/Eurocard/Barclaycard/Visa — Expiry date /

Card No. ☐☐☐☐ ☐☐☐☐ ☐☐☐☐ ☐☐☐☐

☐ Cheque/PO made payable to Database Publications Ltd.

Name _____

Address _____

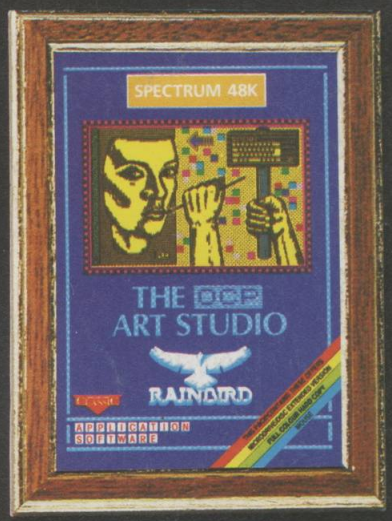
_____

Signed _____

Please allow 28 days for delivery.

MASTER OF THE ART....





AT LAST A SPECTRUM GRAPHICS PACKAGE THAT IS FUN AND EASY TO USE. THE OCP ART STUDIO CONTAINS EVERY FEATURE YOU WILL NEED TO CREATE BEAUTIFUL ILLUSTRATIONS. IT WORKS WITH THE AMX MOUSE FOR EVEN GREATER EASE AND OUR HARD COPY OFFER MEANS YOU CAN HANG YOUR MASTERPIECE ON THE WALL

DO IT ALL – CREATE AN IMAGE. SHRINK IT, EXPAND IT, MOVE IT, ROTATE, COPY IT, COLOUR IT, SPRAY ON A PATTERN OR SHADE. MAKE ELASTIC LINES, TRIANGLES, RECTANGLES, CIRCLES – STRETCH AND MANIPULATE. ADD TEXT OR CHARACTERS, UP, DOWN, SIDEWAYS – ANY SIZE OR PROPORTION. ZOOM IN TO DRAW IN FINE DETAIL. SHRINK THE WHOLE PICTURE TO ADD BACKGROUND.

- Pull down menus. * Icon driven.
- Keyboard, joystick, mouse control.
- Dot matrix printer dumps, 5 sizes and grey scale – up to 80 columns.
- Supports 17 printer interfaces.
- 16 pens, 8 sprays and 16 brushes.
- 32 user-redefinable texture fills.
- Wash texture. * Undo facility.
- Snap facility. * Pixel edit.
- Cut, paste, turn, enlarge, reduce.
- Magnify (3 levels) pan and zoom.
- Text. 9 sizes, 2 directions, bold.
- Font editor with invert, rotate flip, clear, capture from window.
- Elastic line, triangle, rectangle.
- Low cost full colour prints offer.
- Upgrade offer. * Mouse offer.

SINCLAIR USER CLASSIC

"An extremely powerfull utility which should be of use to professional artists and designers as well as the home user"

RAINBIRD

THE OCP ART STUDIO £14.95

OR

EXTENDED ART STUDIO £24.95

For use with disc or microdrive only, and compatible with K DOS and SP DOS disc interfaces (supplied on cassette). Includes disc and microdrive operating systems, screen compression programme, four extra fonts and Kempston mouse compatibility (in addition to AMX). **(Available Mail Order Only)**

FOR 48K ZX SPECTRUM

THE OCP™ ART STUDIO

MAKE CHEQUES OR P.O. PAYABLE TO RAINBIRD SOFTWARE
BARCLAYCARD AND ACCESS ORDERS TEL: 01-240-8837 (24 HOURS)

RAINBIRD SOFTWARE BRITISH TELECOM
WELLINGTON HOUSE UPPER ST MARTIN'S LANE
LONDON WC2H 9DL
TEL: 01-240-8838

RAINBIRD is a division of British Telecommunications plc.