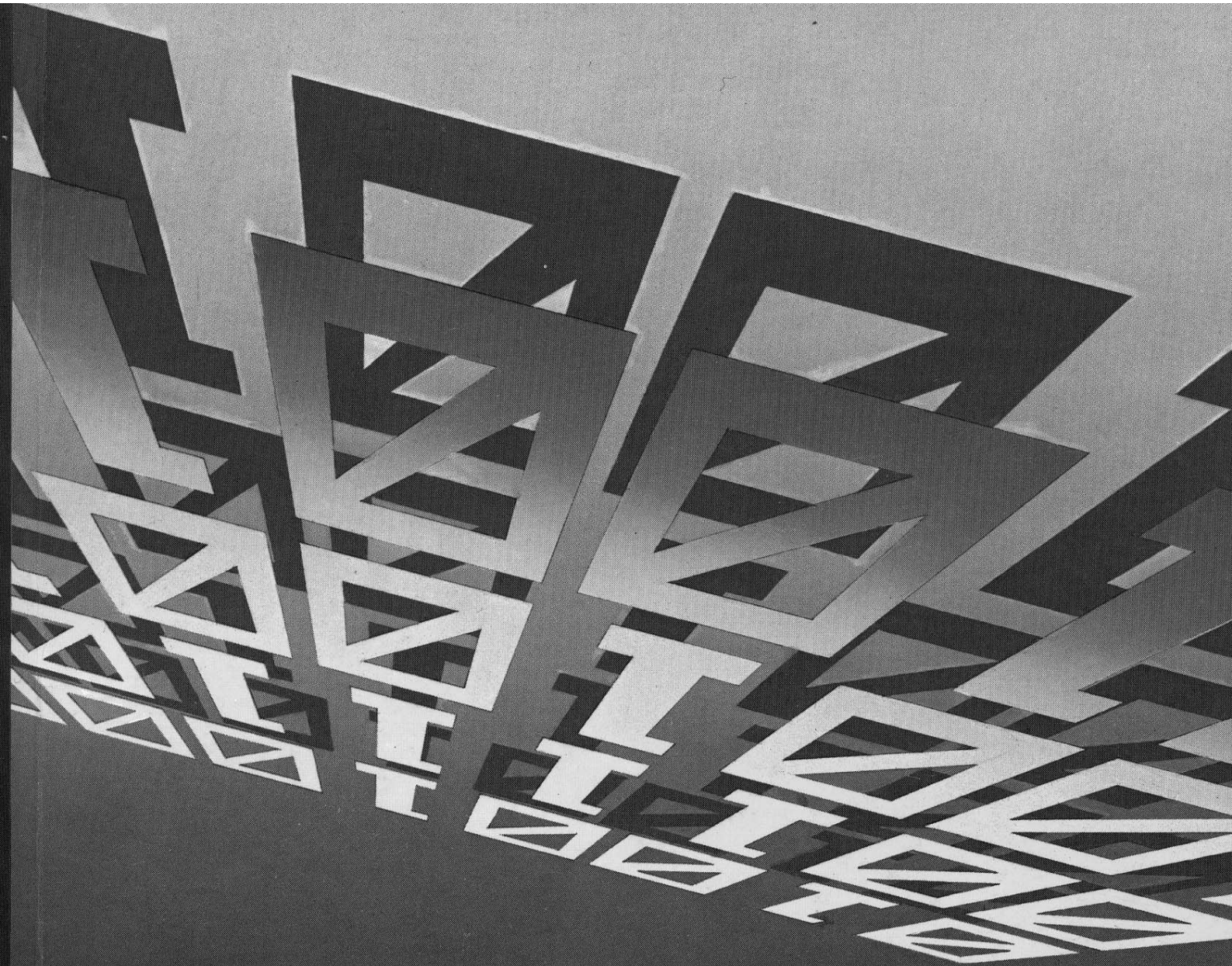


Beyond simple BASIC

DELVING DEEPER

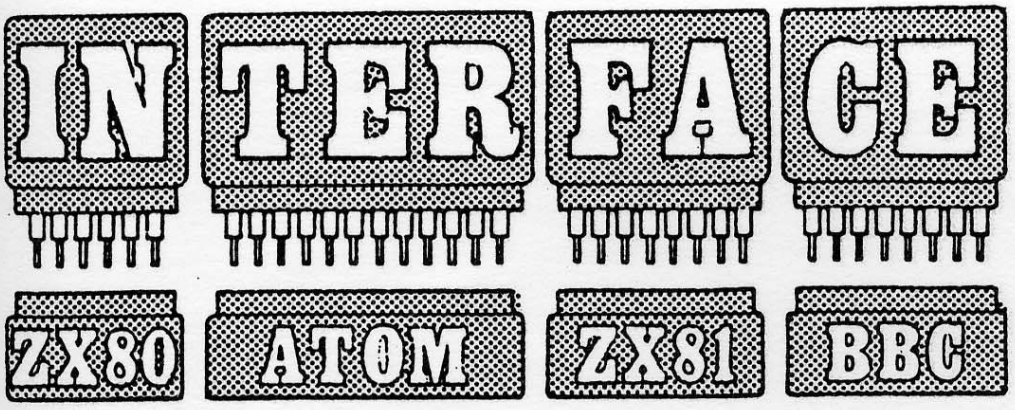
into your

ZX SPECTRUM

Dilwyn Jones

Beyond Simple Basic

DELVING DEEPER into your ZX SPECTRUM



Dilwyn Jones

First published in the UK by:
Interface Publications,
44-46 Earls Court Road,
London, W8 6EJ.

© Copyright 1983, Dilwyn Jones

All rights reserved. This book may not be reproduced in part or in whole without the explicit prior written permission of the publishers. The routines outlined in this book may not be used as part of any program offered for publication nor for programs intended to be sold as software, except as allowed by the publisher. Permission must be sought, in advance, for all applications of this material beyond private use by the purchaser of this volume.

ISBN 0 907563 24 4

Typeset and Printed in England by Commercial Colour Press,
London E7.

CONTENTS

	Page
Introduction	7
Screen tricks	9
Escaping from INPUTs	13
Singulars and plurals	14
CAPS LOCK in programs	15
PAUSE and FOR/NEXT loops	16
Matching up PRINT and PLOT co-ordinates	17
Making a catalogue of programs on a tape	18
Part screen clear	19
Screen scrolling	20
The character set	21
New character set	32
The block graphics	44
Library of subroutines	46
UDG designer program	57
Ready made user defined graphics	63
Sorting out SCREEN$ and ATTR	75
Lower screen attributes	99
Non-deletable program lines	83
"Press Any Key" (IN and OUT)	86
Printing string arrays	95
Preventing AUTO RUNNING	100
Speeding up your programs	101
Making use of the system variables	110
The layout of the Spectrum's memory	132
Other BASICs	145
Understanding the screen display	155
Useful DEF FN calls	161
Input Output Channels	164
Controlling those numbers	166
ROM routines	171

Page

Programs:

Character Designer 57
Stars 76
Noughts and crosses 174
Intruders 178
Super Sounds 182
3D Maze program 194

INTRODUCTION

Welcome, fellow delvers.

Many of the questions and problems I have encountered during my experience with the ZX Spectrum have provided material for this book. The masses of hints and tips contained will help you the programmer to get a deeper understanding of the ZX Spectrum microcomputer — hopefully much more than you ever thought possible.

I hope you find that the fairly advanced material has been written in a simple-to-understand style. The Spectrum is so easy to use compared with other microcomputers that I believe books like this should do everything possible to reflect this friendliness and ease of use.

Whether you want to delve into the ROM or just wander through a maze in glorious 3D effect, I hope you find a lot that interests you.

I would like to thank everyone at Interface Publications both for the opportunity to write this book and for their help and patience during the past months.

Happy Delving,

Dilwyn Jones, Bangor, North Wales, April 1983.

SCREEN TRICKS

Enter and run this program. What does it do?

```
  10 DIM i$(704)
  20 PRINT AT RND*20,RND*31;CHR$
 (RND*223+32)
  30 PRINT AT 0,0; OVER 1; INVER
SE 1;i$
  40 GO TO 20
```

About twice a second something is printed on the screen then the entire screen is inverted. Who needs machine code? Actually, it's done by printing a screenful of spaces OVER the entire screen in INVERSE which has the effect of causing everything that was white on the screen to turn black and everything that was black to turn white; normally you would expect OVER to use its EXOR action to erase some parts, but there isn't anything for EXOR to erase in a string of spaces so it can provide a true screen inversion very quickly. This works well in black and white and it is easy to make it work in colour by adding local PAPER, INK, FLASH and BRIGHT colour controls with an 8 parameter each to prevent global colours playing havoc if they were different. All this does is ensure the same attributes are maintained but that INVERSE 1 is effected.

```
  10 DIM i$(704)
  15 FOR i=1 TO 50
  20 PRINT AT RND*20,RND*31; INK
 RND*7; PAPER RND*7; FLASH RND;
BRIGHT RND;CHR$ (RND*223+32)
  25 NEXT i
  30 PRINT AT 0,0; INVERSE 1; OV
ER 1; PAPER 8; INK 8; BRIGHT 8;
FLASH 8;i$
```

The same idea can be used to turn all text and graphics on the screen a particular colour by omitting the INVERSE 1 statement (or specifying INVERSE Ø) and specifying an INK colour rather than leaving it INK 8. For instance, this program writes random characters on the screen in random INK and PAPER colours, for demonstration, then changes all characters to black while keeping brightness, flashing and PAPER attributes the same:

```
10 DIM i$(704)
15 FOR j=1 TO 50
20 PRINT AT RND*20,RND*31; INK
```

```
 RND*7; PAPER RND*7; FLASH RND;
BRIGHT RND;CHR$ (RND*223+32)
  25 NEXT i
  30 PRINT AT 0,0; OVER 1; PAPER
 8; INK 0; BRIGHT 8; FLASH 8;i$
```

You may have noticed that some INKs and PAPERs come out the same after the random printing in line 20. This is a common problem. Problem? No! Just specify INK 9. You can now read everything.

```
  10 DIM i$(704)
  15 FOR i=1 TO 50
  20 PRINT AT RND*20,RND*31; INK
 RND*7; PAPER RND*7; FLASH RND;
BRIGHT RND;CHR$ (RND*223+32)
  25 NEXT i
  30 PRINT AT 0,0; OVER 1; PAPER
 8; INK 9; BRIGHT 8; FLASH 8;i$
```

We can do the same for the PAPER. By specifying the PAPER colour, and leaving all the other attributes the same, the entire background colour can be changed without disturbing anything on the screen or using CLS. Note that anything written in this colour on the screen may appear to vanish as, say, green text on a green background is not all that easy to read! This example draws random characters with random attributes, then sets the entire background to yellow.

```
  10 DIM i$(704)
  15 FOR i=1 TO 50
  20 PRINT AT RND*20,RND*31; INK
 RND*7; PAPER RND*7; FLASH RND;
BRIGHT RND;CHR$ (RND*223+32)
  25 NEXT i
  30 PRINT AT 0,0; OVER 1; PAPER
 6; INK 8; BRIGHT 8; FLASH 8;i$
```

You can get an interesting effect with any area that has a BRIGHT attribute of 1 with the above program. If you had provided user prompts in BRIGHT 1 or in FLASH 1 (i.e. extra bright or flashing) to highlight them, and then after they had been acted upon you wished to cancel them, you can in the following ways.

To turn off bright spots:

```
  10 DIM i$(704)
  15 FOR i=1 TO 50
  20 PRINT AT RND*20,RND*31; INK
 RND*7; PAPER RND*7; FLASH RND;
BRIGHT RND;CHR$ (RND*223+32)
  25 NEXT i
  30 PRINT AT 0,0; OVER 1; PAPER
 8; INK 8; BRIGHT 0; FLASH 8;i$
```

To turn off flashing:

```
  10 DIM i$(704)
  15 FOR i=1 TO 50
  20 PRINT AT RND*20,RND*31; INK
 RND*7; PAPER RND*7; FLASH RND;
BRIGHT RND;CHR$ (RND*223+32)
  25 NEXT i
  30 PRINT AT 0,0; OVER 1; PAPER
 8; INK 8; BRIGHT 8; FLASH 0;i$
```

Note that in all the above examples, the "screen tricks" are all accomplished in one line! Remember: the answer to the ultimate question of life, the universe and everything is a string of 704 spaces printed OVER 1 over the entire screen in colour 8's!

This technique opens up an interesting possibility — if you want to draw a complex shape which would normally be very slow, first draw it the normal way in the same INK colour as the PAPER colour so that it is invisible, then use the above technique to change the shape's colour so that it almost instantaneously becomes visible. Try this program which draws four concentric circles in magenta on a yellow background. The drawing process takes about 4 seconds.

```
  5 INK 3: PAPER 6: CLS
 10 DIM i$(704)
 15 FOR i=10 TO 70 STEP 20
 20 CIRCLE 120,90,i
 25 NEXT i
```

Try this program which initially draws the circles in yellow on a yellow background then, after drawing, changes the colour of the circles to magenta on yellow. You have to stare at a blank yellow screen for a couple of seconds, but when they appear the circles seem to be drawn almost instantaneously. In prac-

tice, you'd be able to disguise the delay so that the drawing appeared instantaneous.

```
  5 INK 6: PAPER 6: CLS
 10 DIM i$(704)
 15 FOR i=10 TO 70 STEP 20
 20 CIRCLE 120,90,i
 25 NEXT i
 30 PRINT AT 0,0; INK 3; OVER 1
;i$
```

This is only the bare bones of an idea, but using an overprinted string of spaces to control the attributes file is a fast and powerful programming tool.

We've been talking in terms of using a screenful of spaces so far, to affect the whole screen. You can use just enough to alter the attributes of a single character or word on the screen, e.g. to make that green monster turn white with fright when you hit it with your special weapon:

```
PRINT AT Y,X; OVER 1; INK 7;"*"
```

or to make a word cycle through all possible attributes:

```
 10 PRINT AT 5,5;"HELP"
 20 FOR F=0 TO 1: FOR B=0 TO 1:
FOR P=0 TO 7: FOR I=0 TO 7
 30 PRINT AT 5,5; OVER 1; FLASH
F; BRIGHT B; PAPER P; INK I;"
";: REM 4 SPACES
 40 NEXT I: NEXT P: NEXT B: NEX
T F
```

ESCAPING FROM INPUTS

If you wish to stop a program during an INPUT, BREAK does not operate and just causes a space to be added to the INPUT. In the case of a numeric INPUT like INPUT A the answer is to type STOP (which is symbol shift A) followed by ENTER. The program ends with report H STOP in INPUT.

In the case of strings, things are rather different. STOP can still be used, but the cursor must be the first thing in the line and this means moving the cursor out of the quote marks either with DELETE (which is CAPS SHIFT 0) or with CURSOR LEFT (which is CAPS SHIFT 5) then type STOP followed by ENTER and the program will end with report H STOP in INPUT.

Use of the INPUT LINE facility can be problematical in this respect. STOP is accepted as a perfectly valid INPUT character and does not stop the program. However, it is possible to escape from INPUT LINE using CURSOR DOWN (which is CAPS SHIFT 6). You do not need to press ENTER. The program will again come to an end with report H STOP in INPUT even though STOP was not used!

SINGULARS AND PLURALS

A common program bug is to end up with a sentence like "There are 1 bombs left", that is, not making due allowances for singulars and plurals. The program line that generates this sentence may look something like:

```
1000 PRINT "There are ";bombs;"
bombs left"
```

All that is wrong is that the program cannot cope with the grammar of the English language. In this context, it is possible to get around the problem fairly easily. Here is one way around the problem:

```
1000 PRINT "There ";
1010 IF bombs>1 THEN PRINT "are"
;
1020 IF bombs<=1 THEN PRINT "is"
;
1030 PRINT " ";bombs;" bomb";
1040 IF bombs>1 THEN PRINT "s";
1050 PRINT " left"
```

That was a rather long and cumbersome routine for a rather simple task. This routine makes use of AND to shorten it to one program line:

```
1000 PRINT "There ";"are" AND bo
mbs>1;"is" AND bombs<=1;" ";bomb
s;" bomb";"s" AND bombs>1;" left
"
```

TO ALLOW THE SPECTRUM TO SWITCH OFF/ON THE CAPS LOCK

Bit 3 of the system variable 23658 (FLAGS2) indicates the state of CAPS LOCK — if it is engaged bit 3 is 1, if it is not engaged bit 3 is Ø. In programs, it is often useful to be able to detect whether a certain key is pressed irrespective of whether the character produced is upper or lower case, e.g. for a YES or NO answer. To turn on CAPS LOCK use the statement POKE 23658,8 and to turn off the CAPS LOCK use POKE 23658,Ø bearing in mind that these will affect the other flags of the system variable. For a fun demonstration, plug in the ZX printer and enter this:

```
POKE 23658,2
```

You won't damage the printer or waste paper or anything like that, but you will wake it up. So, you can see it is necessary to exercise some caution when POKEing 23658. This is how a YES or NO type of routine might work:

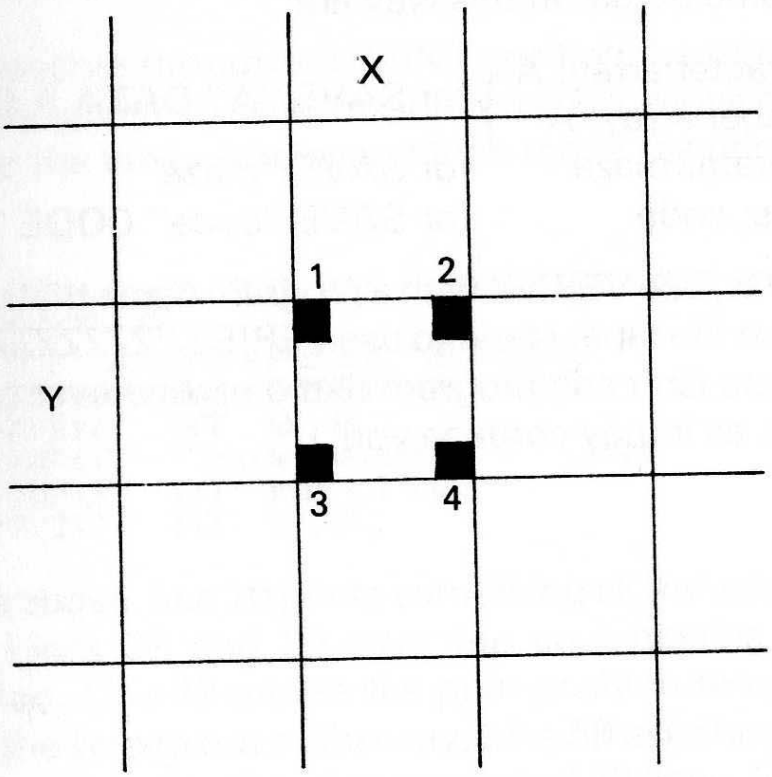
```
1000 PRINT "Do you want another
game (Y or N)?"
1010 POKE 23658,8
1020 IF INKEY$="Y" THEN RUN
1030 IF INKEY$="N" THEN STOP
1040 GO TO 1020
```

PAUSE AND FOR/NEXT LOOPS

There is normally no problem with using PAUSE on the Spectrum but where a fixed delay is needed, PAUSE may cause problems. PAUSE is cut short by a keypress, so if you held your finger on a key all PAUSEs would never occur. This problem can be avoided by the use of FOR/NEXT loops as delay loops. To achieve a delay of approximately 1 second use a loop of FOR A = 1 TO 220 : NEXT A

TO MATCH UP PRINT AND PLOT CO-ORDINATES

Suppose you had a given PRINT AT Y,X; location Y rows down the screen and column X across the screen. Y would be Ø at the top of the screen and 21 at the bottom. X would be Ø at the left of the screen and 31 at the right of the screen — in other words, standard PRINT co-ordinates.



The plot co-ordinates corresponding to the four corners 1 to 4 of character cell Y,X (shaded in the above diagram) would be in PLOT X,Y format:

(1) X * 8, (21 − Y) * 8 + 7
(2) X * 8 + 7, (21 − Y) * 8 + 7
(3) X * 8, (21 − Y) * 8
(4) X * 8 + 7, (21 − Y) * 8

From these you should be able to work out the positions of all pixels within the PRINT position should you need to PLOT or DRAW through a known PRINT position.

MAKING A CATALOGUE OF PROGRAMS ON A TAPE

If you have a tape full of programs and have no knowledge of their names or nature (i.e. BASIC programs, data recordings or bytes of memory) it would obviously be helpful to be able to run through the tape and have the names written on the TV screen along with the type of recording, without affecting what is currently in the computer's memory. The types of recordings you may come across in this way are:

Character array: A Number array: A	} for SAVE "A" DATA A () or A$ ()
Program: maze	for SAVE "maze"
Bytes: code	for SAVE "code" CODE 16384,32

To do this, use VERIFY with a program name that won't ever be found on the tape. I tend to use VERIFY "ZZZZZZZZZZ". Make sure there isn't any program name or whatever on the screen already, as it may confuse you.

PART SCREEN CLEAR

INPUT can be used without having to actually enter a variable. This can be used to clear part of the bottom of the screen as long as INPUT AT does not go past the main PRINT position, (in which case the screen may start scrolling). Use INPUT AT like this, remembering that INPUT AT co-ordinates start from the bottom of the screen.

```
  10 INPUT "Where?",W
  20 FOR A=0 TO 21: PRINT A: NEX
T A
  30 PRINT AT 0,0;
  40 INPUT AT W,0;
```

If INPUT AT reaches the current PRINT position the screen will scroll. You may be able to save the print position, move it out of the way, clear the lower screen and then move the print position back.

```
  10 FOR A=0 TO 21: PRINT AT A,0
;A;: NEXT A
  20 LET X=33-PEEK 23688
  30 LET Y=24-PEEK 23689
  40 PRINT AT 0,0;
  50 INPUT "How many ?",HM
  60 INPUT AT HM+1,0;
  70 PRINT AT Y,X;
```

In the routine above, line 10 prints something all the way down the screen. Lines 20 and 30 save the co-ordinates of the PRINT position. Line 40 moves the print position temporarily to the top of the screen out of the way. Line 50 asks how many lines you want cleared in the upper screen of 22 lines. Entering 1, for example, would only clear line 21. Line 60 does the erasing, and line 70 restores the print position.

SCREEN SCROLLING

It is possible to create a scrolling effect from BASIC by storing the screen image in a string large enough to hold all 22 * 32 characters on the screen. The scrolling is created by rotating the elements of the string. Up and down scrolling can be done in this way quite quickly:

```
 1 REM SCROLL UP
10 DIM A$(704)
20 INPUT A$
30 PRINT AT 0,0;A$
40 LET A$=A$(33 TO )+"
                     "
50 GO TO 30
```

```
 1 REM SCROLL DOWN
10 DIM A$(704)
20 INPUT A$
30 PRINT AT 0,0;A$
40 LET A$="
          "+A$( TO 672)
50 GO TO 30
```

Sideways scrolls are rather slower, but possible:

```
 1 REM SCROLL LEFT
10 DIM A$(704)
20 INPUT A$
30 PRINT AT 0,0;A$
40 FOR F=1 TO 673 STEP 32
50 LET A$(F TO F+31)=A$(F+1 TO
F+31)+" "
60 NEXT F
70 GO TO 30
```

```
 1 REM SCROLL RIGHT
10 DIM A$(704)
20 INPUT A$
30 PRINT AT 0,0;A$
40 FOR F=1 TO 673 STEP 32
50 LET A$(F TO F+31)=" "+A$(F
TO F+30)
60 NEXT F
70 GO TO 30
```

THE CHARACTER SET

In this section we'll make all manner of uses of the ROM character set, then proceed to move the character set out of ROM into RAM (figuratively speaking) and make up our own alphabets. But first, to introduce an important system variable, try this. Enter as a direct command:

POKE 23606,8

then press ENTER of course. Now try typing in a program — if you can! Even more hilarious, try this short typewriter program. Reset the machine and type in:

```
10 POKE 23606,8
20 PRINT INKEY$;
30 IF INKEY$="" THEN POKE 236
06,0: STOP
40 GO TO 20
```

RUN the program and try typing anything — what happens? Letter A's come out as B's, B's as C's and so on. You may like to try this on your local computer-bore then tell him/her the Spectrum's broken. Actually, if you press the SPACE key (without SHIFT) the program resets things to normal. Even funnier, you may like to change line 10 to:

```
10 POKE 23606,4
```

in order to half confuse yourself!

Explanation: 23606/23607 is the address of the system variable that tells the computer where the data lists required for the display of characters on the screen lie. This two byte system variable contains a number, 15360, after switch-on which is 256 less than where this table starts in the ROM. So 15360 + 256 = 15616. This is where the ROM CHARACTER GENERATOR starts. Let's have a PEEK in there to see what's there. ENTER and RUN this program. You should get similar results to the sample run that follows.

```
10 FOR A=15616 TO 15639
20 PRINT PEEK A
30 NEXT A
```

```
0
0
0
0
```

```
0
0
0
0
0
16
16
16
16
0
16
0
0
36
36
0
0
0
0
0
```

That wasn't particularly informative was it? The 15639 wasn't significant — just a number chosen to give a short example printout. Even if we look up the character set in the manual it doesn't seem to have any relevance to the numbers printed out. Ah well, another scratch of the old head is required.

Let us try in binary, since computers are said to make a lot of use of that. Type in and RUN this program, remembering the two apostrophes in line 70 which are easily forgotten.

```
20 LET P=PEEK A
30 FOR B=7 TO 0 STEP -1
40 PRINT AT 21,B;P-2*INT (P/2)
50 LET P=INT (P/2)
60 NEXT B
70 PRINT AT 21,12;A''
80 NEXT A
```

```
00000000     15616
00000000     15617
00000000     15618
00000000     15619
00000000     15620
00000000     15621
00000000     15622
00000000     15623
00000000     15624
00010000     15625
00010000     15626
00010000     15627
```

```
00010000      15628
00000000      15629
00010000      15630
00000000      15631
00000000      15632
00100100      15633
00100100      15634
00000000      15635
00000000      15636
00000000      15637
00000000      15638
00000000      15639
```

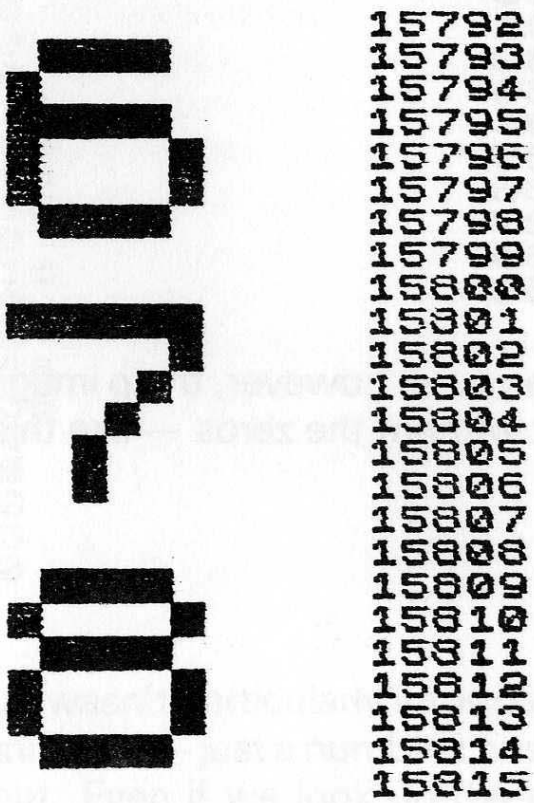
This may not look any better to you — however, try to imagine the left column of the printout without the zeros — like this:

```
              15616
              15617
              15618
              15619
              15620
              15621
              15622
              15623
              15624
  1           15625
  1           15626
  1           15627
  1           15628
              15629
  1           15630
              15631
              15632
1   1         15633
1   1         15634
              15635
              15636
              15637
              15638
              15639
```

What you should see is a SPACE, an exclamation mark and a quote symbol (if you're imaginative). For our next trick we'll scan the entire character generator so that we can see what is contained in its dark depths. To ensure a better screen picture we'll use ■ instead of ones.

```
  10 FOR A=15616 TO 16383
  20 LET P=PEEK A
  30 FOR B=7 TO 0 STEP -1
  40 PRINT AT 21,B;"■" AND (P-2*
INT (P/2))=1
  50 LET P=INT (P/2)
  60 NEXT B
  70 PRINT AT 21,12;A''
  80 NEXT A
```

Here's a sample of what you should get.



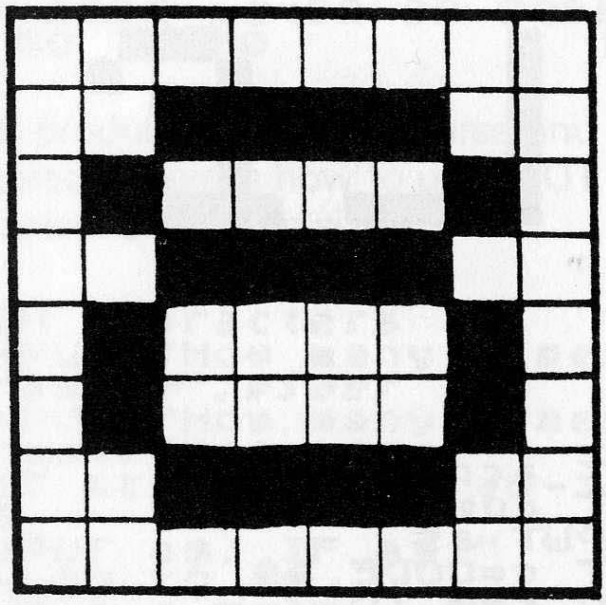
If you have a printer and would like to keep a copy on paper, you could use this program which will give you an enlarged printout of all characters stored in the ROM character generator. Warning: you'll need a lot of paper for this! Press BREAK to stop the printout when you've had enough.

```
   10 FOR A=15616 TO 16383
   20 LET P=PEEK A
   30 DIM A$(8)
   40 FOR B=8 TO 1 STEP -1
   50 IF P-INT (P/2)*2 THEN LET A
$(B)="■"
   60 LET P=INT (P/2)
   70 NEXT B
   80 LPRINT A$;TAB 10;A;TAB 18;P
EEK A
   90 NEXT A
```

You should by now be getting the idea that the character generator holds a bit-by-bit pattern of what the characters look like on the screen. Also, if you refer to Appendix A of the Spectrum manual you'll see that the characters come up in the same order on screen as they do in the appendix — at least those in the range 32 to 127. The others are either CONTROL CHARACTERS (they control the screen in some way or other, rather than appear on it), graphics characters (which are stored

elsewhere), block graphics (which are "calculated" rather than stored as bit patterns) or are made up of combinations of characters in the character generator (e.g. RUN, INKEY$, IF, etc.) in which case there is another table in the ROM that indicates which combinations.

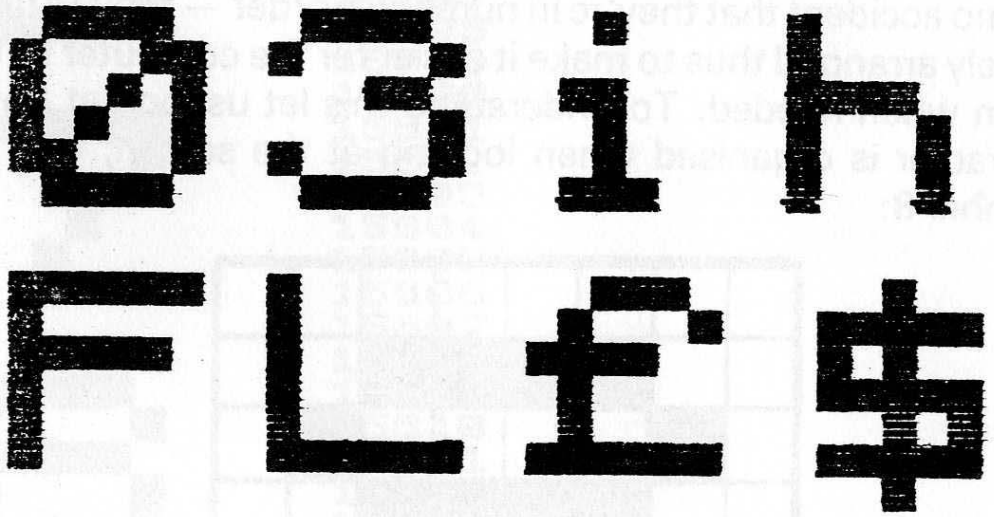
It is no accident that they're in numerical order — they're deliberately arranged thus to make it easier for the computer to find them when needed. To understand this let us look at how a character is organised when looking at the screen, e.g. the number 8:



A character is made up of eight by eight pixels (picture cells or little squares if you like) on the screen. By luck there just happens to be eight bits per byte (would you believe it???). So, if every byte of eight bits represented a row across of the character, we could store a pattern of the character in eight bytes. This is how the character generator works: there are eight bytes for each character storing the pattern as a series of zeros and ones. The ones represent the parts of the characters inked in and the zeros represent the parts of the character left blank (in paper colour). This is how the bit pattern of the figure 8 would look (the left column is the bit pattern, the right column shows where in the ROM in decimal that it's located):

```
00000000   15808   0
00111100   15809   60
01000010   15810   66
00111100   15811   60
01000010   15812   66
01000010   15813   66
00111100   15814   60
00000000   15815   0
```

One use we can put this to is for enlarging the display characters. If we printed a ■ for every 1 we could enlarge the characters eight times! Let's try this. The program is slightly different from those we've used so far so study it carefully.



```
 10 LET across=0
 20 LET down=0
 30 INPUT a$
 40 LET c=CODE a$
 50 FOR k=0 TO 7
 60 LET p=PEEK (15360+c*8+k)
 70 FOR f=0 TO 7
 80 PRINT AT down+k,across+7-f;
"■" AND (p-2*INT (p/2)=1)
 90 LET p=INT (p/2)
100 NEXT f
110 NEXT k
120 IF across+8>31 THEN LET dow
n=down+8
130 LET across=across+8 AND acr
oss+8<=31
140 GO TO 30
```

The two variables *down* and *across* control where on the screen that printing takes places. A$ is the character you enter to be enlarged. This should consist of one character with a CODE of 32 to 127 (i.e. SPACE to the © copyright symbol). c is the CODE of this character. Note in line 60 how the number 15360 is used for the bottom of the character set table. You may remember that this number is 256 less than the address of the start of the table. Why?

Well, the first pattern in the table is that of SPACE, which is CHR$(32). Remember also that there are eight bytes for the dot patterns of every character, so everything starts at multiples of eight. 8 times 32 happens to be 256 and 15360 + 256 is 15616, the start of the table in the ROM. The loops carry on dividing by two and finding the remainder each time to determine whether or not to shade in a part of the screen. The values of *down* and *across* are then adjusted. You may like to enter this additional line to prevent characters that the program can't handle being entered (you are made to re- enter any characters rejected).

```
  35 IF CODE a$<32 OR CODE a$>12
7 THEN GO TO 30
```

The characters produced are very coarse and you cannot fit many on the screen. Here is how to use PLOT and DRAW to generate characters of varying sizes.

```
   1 REM characters
  10 INPUT "How many times wider
 (1=normal)?";wider
  20 INPUT "How many times talle
r(1=normal)?";taller
  30 LET across=wider*8-1: LET d
own=176
  40 INPUT a$: IF a$<" " OR a$>"
@" THEN GO TO 40
  50 FOR a=0 TO 7
  60 LET peek=PEEK (15360+CODE a
$*8+a)
  70 FOR b=0 TO 7
  80 IF peek-2*INT (peek/2) THEN
 FOR t=1 TO taller: PLOT across-
b*wider,down-a*taller-t: DRAW 1-
wider,0: NEXT t
 100 LET peek=INT (peek/2): NEXT
 b
 110 NEXT a
 120 LET across=across+wider*8
 130 IF across>255 AND down-tall
er*8>taller*8-1 THEN LET down=do
wn-taller*8: LET across=wider*8-
1
 140 IF across>255 AND down-tall
er*8<taller*8 THEN PRINT AT 21,3
1;: FOR a=1 TO taller: PRINT : N
EXT a: LET across=wider*8-1
 150 GO TO 40
```

The program is complicated so study the following information carefully. When you run the program you will be asked first how many times wider than normal you want the character to

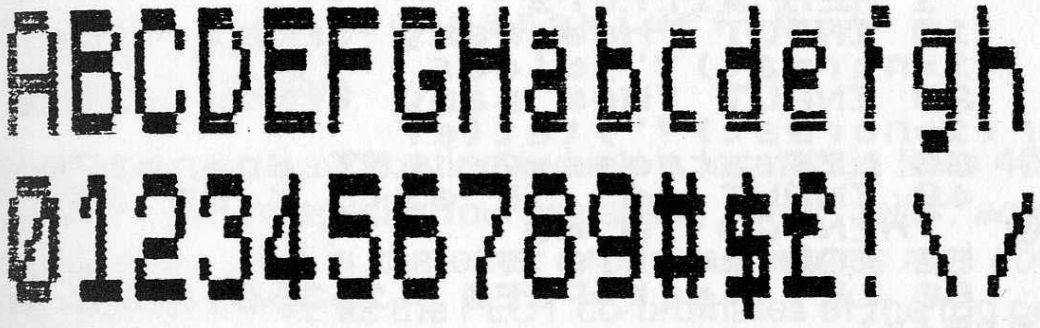
appear on screen. If you want the characters three times as wide as normal, then you should type 3, followed by ENTER. If you want normal width characters enter 1. You should do the same when asked how many times taller than normal you wanted. The program will start at the top left of the screen and work its way across and down the screen until it reaches the bottom, when the whole lot will scroll upwards to make room for another line. The program runs in black and white (or the permanent INK and PAPER colours you set up) but you may wish to add suitable colour statements of your own in the program.

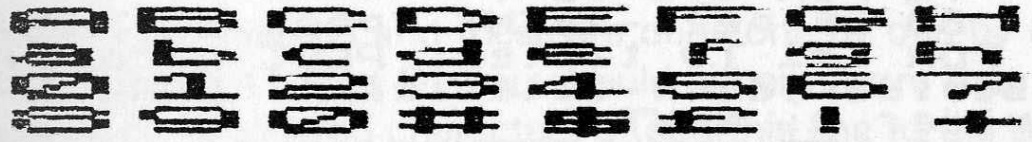
The variables *taller* and *wider* are used in their full form throughout to make their meanings clearer. The same is true of the variable *peek* (it's written in lower case in the listing to distinguish it from the keyword PEEK) and variables *across* and *down*. These are co-ordinates for the PLOT commands used later. 176 is one greater than the limit value of 175 for PLOT — don't worry about generating errors though. *across* starts off a certain amount along the screen because all plotting and drawing is done from right to left due to the way the binary is evaluated. Line 40 scans to see which key you're pressing. This program uses INPUT to see which letters or symbols you want to enlarge since there are a few symbols not easily obtainable with INKEY$, which might otherwise save you the bother of pressing ENTER all the time. Line 50 starts the loop which looks in the character generator for the eight bytes needed to assemble the character on screen. These bytes are found by line 60. The loop starting at line 70 determines the pattern on its screen by taking the byte from the character generator and repeatedly checks to see if individual bits are set so that appropriate areas may be shaded in on the screen. This is done by DRAWing a line which is *wider* times longer than a pixel. This is done *taller* times to make it the right height. Line 100 divides the value of *peek* by 2 so that the next bit may be checked. Line 120 sets the new value of *across* — if this goes off the right-hand side of the screen, a new value is given to *down* and the program decides if it is necessary to scroll, which is done by using PRINT enough times. *across* is reset to its initial value for a new line of characters. After all that, the keyboard is scanned again and the whole story is repeated.

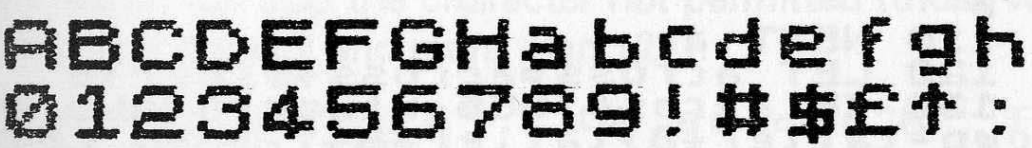
The program works best with integer values of *wider* and *taller*, but you can have good results for non-integer values of *wider* — even creating the illusion of having more or less characters per line by fiddling with the value of *across*:

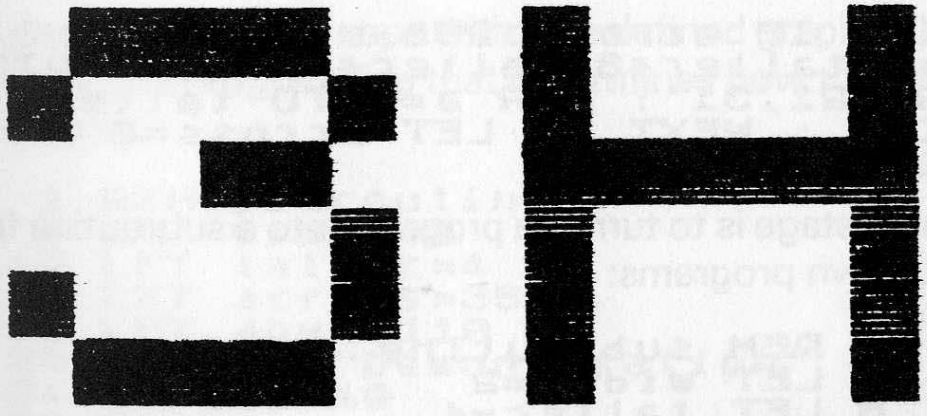
$$Across = \frac{\text{no. of characters normally across the screen}}{\text{no. of characters required across the screen}}$$

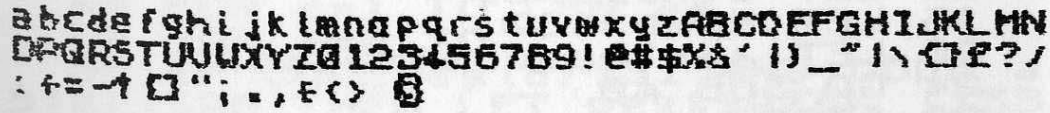
e.g., for forty characters per line, *across* would be 32/40 or 0.8. Here are some example displays produced by the program including a 40 character per line display.



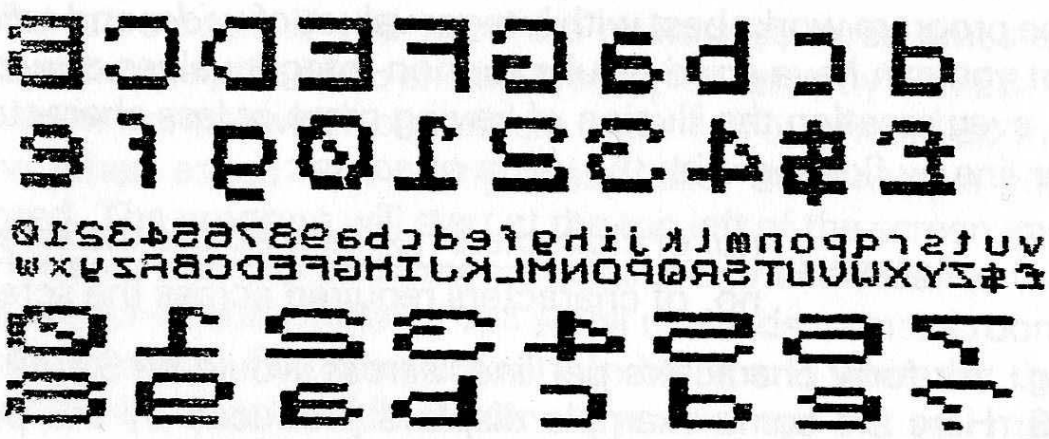








Using the same principle we can use the program to create mirror image characters e.g.:



This is done by simply drawing in the opposite direction.

```
  1 REM mirrors
 10 INPUT "How many times wider
(1=normal)?";wider
 20 INPUT "How many times talle
r(1=normal)?";taller
 30 LET across=0: LET down=176
 40 INPUT a$: IF a$<" " OR a$>"
©" THEN GO TO 40
 50 FOR a=0 TO 7
 60 LET peek=PEEK (15360+CODE a
$*8+a)
 70 FOR b=0 TO 7
 80 IF peek-2*INT (peek/2) THEN
 FOR t=1 TO taller: PLOT across+
b*wider,down-a*taller-t: DRAW 1-
wider,0: NEXT t
 100 LET peek=INT (peek/2): NEXT
 b
 110 NEXT a
 120 LET across=across+wider*8
 130 IF across>256-wider*8 AND d
own-taller*8>taller*8-1 THEN LET
 down=down-taller*8: LET across=
0
 140 IF across>256-wider*8 AND d
own-taller*8<taller*8 THEN PRINT
 AT 21,31': FOR a=1 TO taller: P
RINT : NEXT a: LET across=0
 150 GO TO 40
```

The next stage is to turn the program into a subroutine for use in your own programs:

```
  1 REM subroutine
  4 LET wider=2
  8 LET taller=4
 12 LET across=35
 16 LET down=110
 20 LET a$="Demonstration"
 24 GO SUB 40
 30 STOP
 40 FOR d=1 TO LEN a$
 50 FOR a=0 TO 7
 60 LET peek=PEEK (15360+CODE a
$(d)*8+a)
```

```
  70 FOR b=0 TO 7
  80 IF peek-2*INT (peek/2) THEN
 FOR t=1 TO taller: PLOT across-
b*wider,down-a*taller-t: DRAW 1-
wider,0: NEXT t
  90 LET peek=INT (peek/2): NEXT
 b
 100 NEXT a
 110 LET across=across+wider*8
 120 IF across>255 AND down-tall
er*8>taller*8-1 THEN LET down=do
wn-taller*8: LET across=wider*8-
1
 130 IF across>255 AND down-tall
er*8<taller*8 THEN PRINT AT 21,3
1;: FOR a=1 TO taller: PRINT : N
EXT a: LET across=wider*8-1
 140 NEXT d
 150 RETURN
```

Before calling the subroutine (which is located in lines 40 to 160) you need to specify four variables, *wider*, *taller*, *across* and *down* as used in the other programs. *across* and *down* need to be specified as the PLOT co-ordinates of the top right corner of the first character. A$ is a string containing the characters to be enlarged and these are done one by one by the extra FOR/ NEXT loop D. You should either ensure that A$ contains no unallowed characters or add this line to the program which will skip the character not permitted (those with CODEs less than 32 and more than 127):

```
  45 IF a$(d)<" " OR a$(d)>"©" T
HEN GO TO 150
```

The subroutine can also cope with user defined graphics by the addition of one program line that determines how the value of *peek* is derived.

```
   1 REM subroutine
   4 LET wider=2
   8 LET taller=4
  12 LET across=35
  16 LET down=110
  20 LET a$="Demonstration"
  24 GO SUB 40
  30 STOP
  40 FOR d=1 TO LEN a$
  50 FOR a=0 TO 7
  60 IF CODE a$(d)>31 AND CODE a
$(d)<128 THEN LET peek=PEEK (153
60+CODE a$(d)*8+a)
  61 IF CODE a$(d)>143 AND CODE
a$(d)<165 THEN LET peek=PEEK (US
R "a"+(CODE a$(d)-144)*8+a)
  70 FOR b=0 TO 7
```

```
  60 IF peek-2*INT (peek/2) THEN
 FOR t=1 TO taller: PLOT across-
b*wider,down-a*taller-t: DRAW 1-
wider,0: NEXT t
  90 LET peek=INT (peek/2): NEXT
 b
 100 NEXT a
 110 LET across=across+wider*8
 120 IF across>255 AND down-tall
er*8>taller*8-1 THEN LET down=do
wn-taller*8: LET across=wider*8-
1
 130 IF across>255 AND down-tall
er*8<taller*8 THEN PRINT AT 21,3
1': FOR a=1 TO taller: PRINT : N
EXT a: LET across=wider*8-1
 140 NEXT d
 150 RETURN
```

Our next project will be to set up a new character set in RAM that can be used to write on screen, produce listings and so on — in fact, be identical in all respects but appearance. The Spectrum manual hints that this can be done, but offers little in the way of explanation how. The new character set will be located above RAMTOP, below the user defined graphics characters. A step by step guide as to how to do this and how to ultimately redefine any character you like at will, will be given. It all depends on the fact that system variables 23606/7 point to the start of the character set. Where possible, addresses are given to suit 16K RAM and 48K RAM Spectrums unless there is a method of calculating addresses for machines with any amount of memory. The new character set will be a right-sloping italic-style text with this type of appearance:

```
  This is a sample printout to
show the NEW CHARACTER SET in
action. Only the letters and the
numbers 0123456789 have been
redefined, although you will be
shown later how to redefine any
character you wish.
```

```
for now, symbols such as $%£!%
remain unchanged although they
may be redefined later if
required. There follows a sample
listing produced with this
character set.


The listing is for a simple
typewriter. It has a delete
facility to erase the last
character: press DELETE (Caps
Shift 0). Press ENTER to start a
new line of text.


  5 BORDER 0
 10 PRINT INKEY$;
 20 IF INKEY$<>"" THEN GO TO 20
 30 IF INKEY$="" THEN GO TO 30
 34 BEEP .01,25
 37 IF INKEY$=CHR$ 12 THEN PRIN
T CHR$ 8;" ";CHR$ 8;: GO TO 20
 40 GO TO 10
```

Step 1 The first step is to bring RAMTOP down by 768 bytes (the number of bytes in the character generator) to make room between the end of BASIC and the user defined graphics in which to store the new character set. There are 768 bytes in the new character set like the ROM version.

16K Spectrums: The new RAMTOP needed will be 31831 as opposed to the normal value of 32599. To lower RAMTOP to this new value enter the command CLEAR 31831.

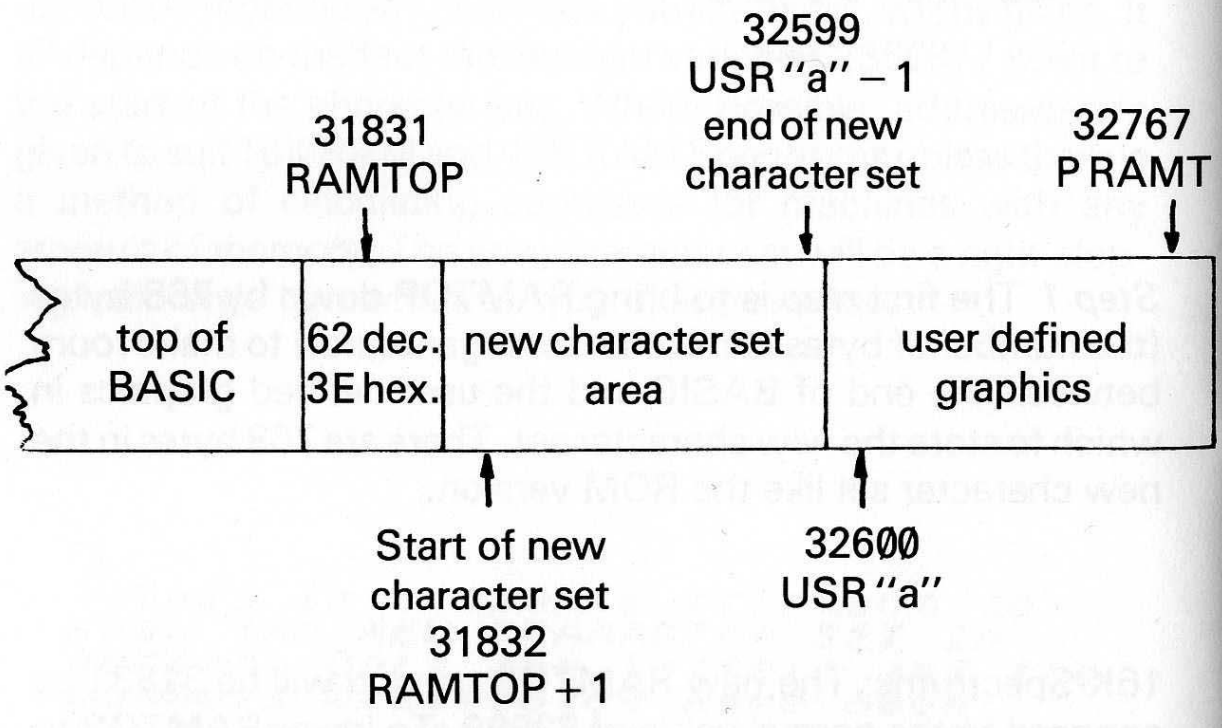
48K Spectrums: The new RAMTOP required will be 64599 as opposed to the normal value of 65367. To lower RAMTOP to this new value enter the command CLEAR 64599.

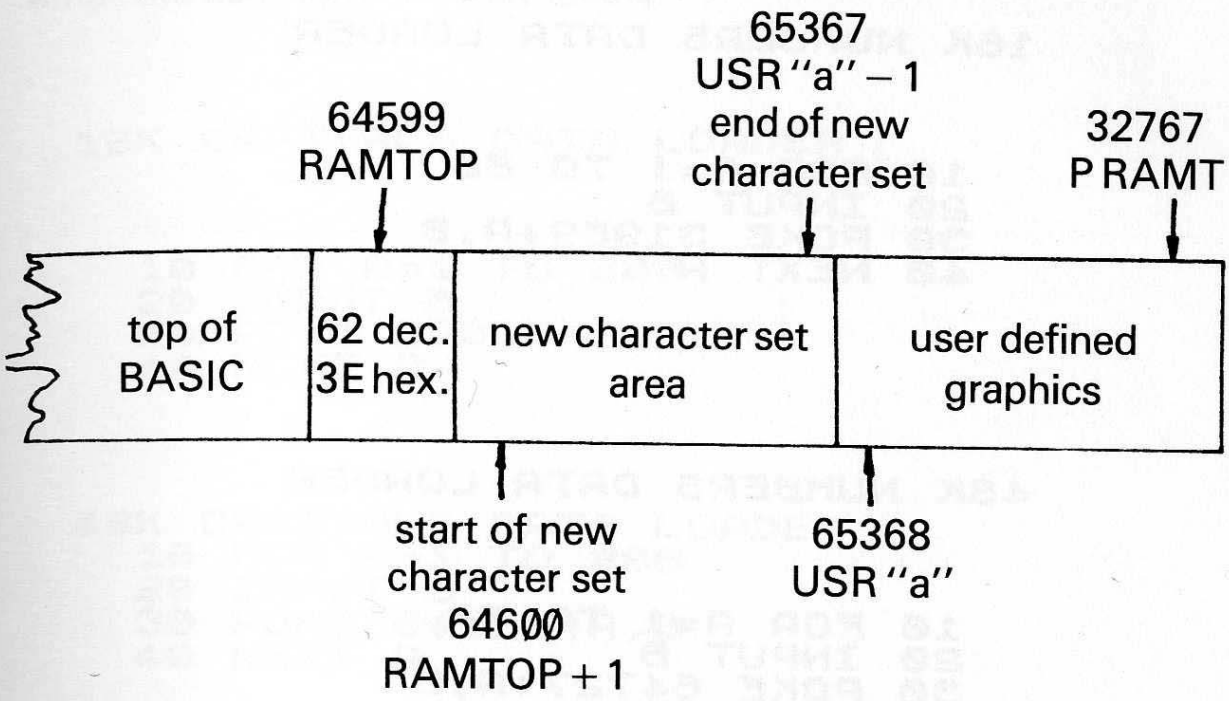
Both these versions of the command rely on absolute numbers. If you have a different amount of memory connected, you will need to replace the numbers with an expression that allows the correct values to be worked out to suit the circumstances. This could also be the case if you had anything else like machine code routines stored above RAMTOP. This expression will lower RAMTOP from its current address by 768 bytes every time it is used by looking up the address of RAMTOP in system variable 23730/1, subtracting 768 from this and use CLEAR with this value like this:

```
CLEAR (PEEK 23730+256*PEEK 2373
1-768)
```

If suitable addresses are not given for the 48K machines in the following pages, the 16K values are easily converted by adding 32768 in most cases, except of course for system variables. This is a diagram of how the memory is now laid out:

16K Spectrums





Step 2 Now the existing character set must be copied up from ROM to this area of RAM so that those we do not wish to reprogram will be the same as they always were.

16K Spectrum version:

```
16K CHARACTER SET COPIER


10 FOR A=15616 TO 16383
20 POKE 16216+A,PEEK A
30 NEXT A
```

48K Spectrum version:

```
48K CHARACTER SET COPIER


10 FOR A=15616 TO 16383
20 POKE 48984+A,PEEK A
30 NEXT A
```

It is important that this is entered correctly as any mistakes will be difficult to correct later on and you may have to start again from scratch.

Step 3 Now to start redefining. I will tell you first how to redefine numbers, capital letters and lower case letters to right-sloping italic-style face.

First of all, let us redefine the numbers. Enter this program:

16K NUMBERS DATA LOADER

```
10 FOR A=1 TO 80
20 INPUT B
30 POKE 31959+A,B
40 NEXT A
```

48K NUMBERS DATA LOADER

```
10 FOR A=1 TO 80
20 INPUT B
30 POKE 64727+A,B
40 NEXT A
```

Enter these numbers when you are running the above program. Enter the top row first, left to right, then the second row and so on (just in case you enter the zeros first and waste your energy....).

DATA FOR NUMBERS

```
0   60   70   74   148  164  120  0
0   48   80   16   32   32   248  0
0   28   34   4    56   64   124  0
0   30   4    24   4    72   56   0
0   6    10   20   36   126  8    0
0   30   16   60   2    68   56   0
0   30   32   60   66   68   56   0
0   30   2    4    8    16   32   0
0   28   36   56   68   68   56   0
0   28   34   34   28   4    56   0
```

Note that you won't see any effect yet — we'll change a certain system variable later.

Step 4 Now to redefine the capital letters. Use this program to enter the 208 numbers that follow!

```
16K CAPITALS DATA LOADER

   10 FOR A=1 TO 208
   20 INPUT B
   30 POKE 32095+A,B
   40 NEXT A


48K CAPITALS DATA LOADER
   10 FOR A=1 TO 208
   20 INPUT B
   30 POKE 64863+A,B
   40 NEXT A


DATA FOR CAPITALS

0     12    16    34    62    66    66    0
0     28    16    60    34    66    124   0
0     28    34    32    64    68    56    0
0     24    20    34    34    68    120   0
0     30    16    60    32    64    120   0
0     30    16    60    32    64    64    0
0     28    34    32    76    68    56    0
0     18    18    60    36    72    72    0
0     62    8     16    16    32    248   0
0     2     2     4     68    72    56    0
0     18    20    56    40    68    66    0
0     16    16    32    32    64    124   0
0     34    54    42    66    68    68    0
0     18    26    42    44    68    68    0
0     28    34    34    68    68    56    0
0     28    18    34    60    64    64    0
0     60    66    66    164   148   120   0
0     28    18    34    60    68    66    0
0     28    34    24    4     68    56    0
0     62    8     8     16    16    32    0
0     17    34    34    68    68    56    0
0     34    34    36    36    40    16    0
0     33    33    66    66    90    36    0
0     34    20    24    56    68    130   0
0     34    20    8     16    32    64    0
0     62    4     8     16    32    124   0
```

Step 5 Now to redefine the lower case letters. Use this program to enter the following numbers:

```
16K LOWER CASE DATA LOADER

  10 FOR A=1 TO 208
  20 INPUT B
  30 POKE 32351+A,B
  40 NEXT A

48K LOWER CASE DATA LOADER

  10 FOR A=1 TO 208
  20 INPUT B
  30 POKE 65119+A,B
  40 NEXT A

DATA FOR LOWER CASE

0    0   12   2   60   68   56   0
0   16   16  60   34   66  124   0
0    0   28  32   32   64   56   0
0    2    2  28   36   68   56   0
0    0   28  34  124   64   56   0
0    6    8  12   16   16   32   0
0    0   14  18   34   60    4  120
0   16   16  62   34   68   68   0
0    4    0  24    8   16  120   0
0    2    0   4    4    8   72   48
0   16   20  56   48   72   68   0
0    8   16  16   32   32   24   0
0    0   54  73   73  146  146   0
0    0   60  34   34   68   68   0
0    0   28  34   36   68   56   0
0    0   28  18   34   60   64   64
0    0   30  18   36   60    8   30
0    0   14  16   16   32   32   0
0    0   30  32   24    4  120   0
0    4   30   8   16   18   12   0
0    0   34  34   68   68   56   0
0    0   34  34   36   40   16   0
0    0   65  65  146  146  106   0
0    0   34  20   24   40   68   0
0    0   18  36   60    8   16   96
0    0   60   8   16   32  120   0
```

Step 6 Using it. You'll be glad to know we've nearly finished typing. To use this new character set we have to alter the value of system variable 23606/7 CHARS (see Spectrum manual ch.25) which is the pointer to the start of the character set. It normally has a value of 256 less than the start of the first byte of the character generator to make it easy to find the address of the data for each character. How? It merely allows a simple mathematical manipulation of CODEs, meaning that the start address of individual characters can be found from the value

held in 23606/7 and the CODE of those characters multiplied by 8; 23606/7 normally have the values 0 and 60 respectively when using the ROM character set on both 16K and 48K Spectrums, i.e.:

PEEK 23606 is 0
PEEK 23607 is 60 } 0 + 256 * 60 is 15360

To make this point to the new character set, on a 16K Spectrum we'd use:

POKE 23606,88: POKE 23607,123 (88 + 256*123 = 31576)

On a 48K Spectrum this would be:

POKE 23606,88:POKE 23607,251 (88 + 256*251 = 64344)

```
16K VERSION

POKE 23606,88: POKE 23607,123

48K VERSION

POKE 23606,88: POKE 23607,251
```

It is best to enter both as one long direct command joined by a colon in the normal way because entered as individual commands, they don't wait for you and you get garble on the screen (meaning that you don't see what you're doing entering the other one). If everything's been done right anything you now type appears in the new lettering, although anything that was previously on the screen remains as it was. The values POKEd gave 23606/7 a value of 31576 for 16K users and 64344 for 48K users. To change back to the ROM character set on both 16K and 48K machines use:

POKE 23606,0: POKE 23607,60

```
POKE 23606,0: POKE 23607,60
```

You can use both as program statements so you can change back and forth between both character sets during a program if you like. Since anything printed on the screen stays there unless overwritten or cleared, you could freely mix both typefaces on the screen if you like. You can also make printer listings in sloping characters if you like. The sloping character set

has a big advantage on a ZX printer since using the normal vertical set shows up any deficiencies in the quality of the printout — the new sloping character set helps hide this. In fact, the appearance is the only thing that leads to a problem. SCREEN$ identifies characters on screen by looking up in the character generator for a matching character from which it can tell which character it is. If you're using the new character set then ask SCREEN$ to identify a character printed with the old ROM character set, it will not be identified and returns the empty string. SCREEN$ must look up in the same character set from which the character was originally written on screen. This is only a problem if you're chopping and changing between two or more character sets!

Step 7 Saving it on tape and reloading. Before you do anything else, save the new character set on tape so that you can load it back into the computer when you need it again. I'm glad to say it's easier to load and use than it was to enter and set up.

To save the new character set on tape from a 16K Spectrum:

SAVE "chars" CODE 31832,768

```
SAVE "chars"CODE 31832,768
```

To save the new character set on tape from a 48K Spectrum:

SAVE "chars" CODE 64600,768

```
SAVE "chars"CODE 64600,768
```

To load the new character set from tape back into your 16K Spectrum, type and enter:

CLEAR 31831: LOAD "chars"CODE 31832,768

```
 CLEAR 31831: LOAD "chars"CODE 3
1832,768
```

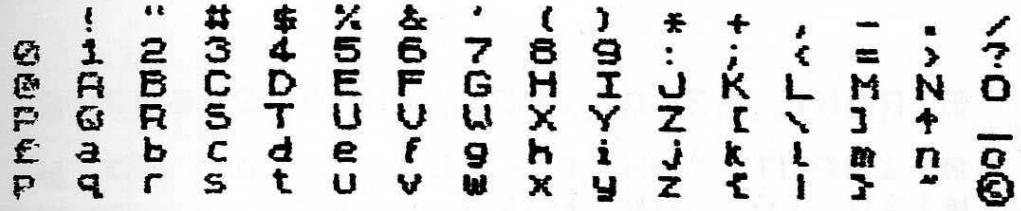
On a 48K machine:

CLEAR 64599: LOAD "chars"CODE 64600,768

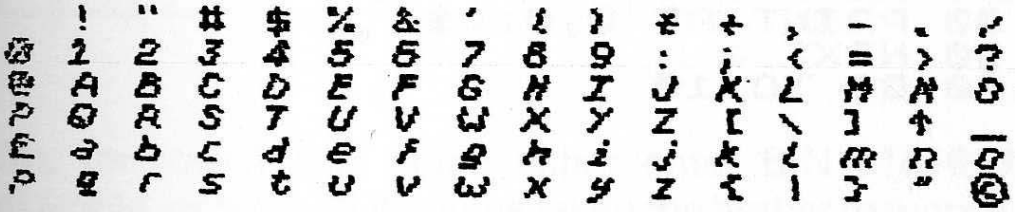
```
 CLEAR 64599: LOAD "chars"CODE 6
4600,768
```

Remember to VERIFY after SAVEing the character set because you'll have a lot of retyping to do if anything went wrong! Incidentally, LOAD ""CODE will reload the character set back into the same place as it used to be — useful to save a bit of typing. This is what the new character set looks like:

Normal ROM character set:



Alternative RAM character set:



Except for SCREEN$'s restrictions both sets work exactly the same. Functions, keywords, etc., are expanded using whichever character set is in force at the time. Only the letters and numbers have been redefined so, if you wish to redefine individual characters rather than do a job lot like that described, here is how this can be done. Go as far as *Step 2* above, then enter this program. Two versions, one for 16K, the other for 48K:

```
16K REDEFINE ANY CHARACTER
   5 POKE 23606,88: POKE 23607,1
23
  10 INPUT "Which character do y
ou wish to  define? ";c$
  20 IF c$="" THEN STOP
```

```
 30 IF c$<" " OR c$>"©" THEN GO
TO 10
 40 LET c=CODE c$
 50 FOR a=0 TO 7
 60 INPUT ("Which value for row
";a+1;" ? ");value
 70 POKE 31576+c*8+a,value
 80 PRINT AT 0,0;c$
 90 NEXT a
100 GO TO 10
```

48K REDEFINE ANY CHARACTER

```
  5 POKE 23606,88: POKE 23607,2
51
 10 INPUT "Which character do y
ou wish to  define? ";c$
 20 IF c$="" THEN STOP
 30 IF c$<" " OR c$>"©" THEN GO
TO 10
 40 LET c=CODE c$
 50 FOR a=0 TO 7
 60 INPUT ("Which value for row
";a+1;" ? ");value
 70 POKE 64344+c*8+a,value
 80 PRINT AT 0,0;c$
 90 NEXT a
100 GO TO 10
```

Once you have copied the character set from ROM into RAM, RUN this program. Line 10 asks you which character you wish to change. For instance, if you wanted to change the bracket symbol (into a sloping style, just enter (.

If you want to stop the program, just press ENTER by itself — the program will STOP if you just press ENTER to enter a null string. Line 30 restricts the characters that may be redefined to those from SPACE to the © copyright symbol (CHR$ 32 to 127).

c is the CODE of the character to be redefined. This is used to determine where the character lies (its address). The loop in line 50 allows you to change all eight bytes of the character's dot pattern in RAM. Line 60 asks you to enter the new value for the eight rows of the dot patterns. This could

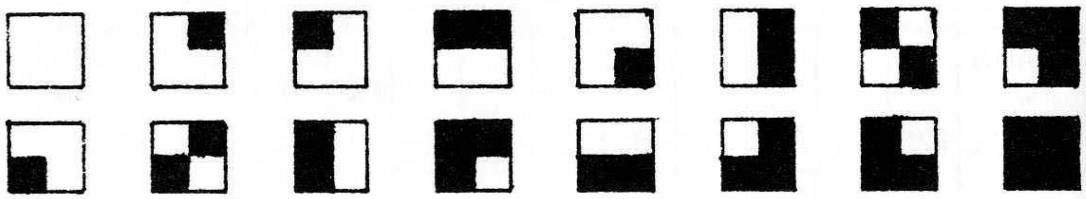
be done in two ways. You must treat these as you would the USER DEFINED GRAPHICS — write out their bit patterns, e.g.:

VALUE / ROW	128	64	32	16	8	4	2	1	
1	Ø	1	Ø	1	Ø	1	Ø	1	64 + 16 + 4 + 1 = 85
2	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	= Ø
3	1	Ø	Ø	Ø	Ø	1	Ø	Ø	128 + 4 = 132
4	Ø	Ø	Ø	Ø	Ø	Ø	Ø	1	= 1
5	Ø	Ø	Ø	Ø	Ø	Ø	1	1	2 + 1 = 3
6	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	= Ø
7	Ø	1	1	1	Ø	Ø	Ø	Ø	64 + 32 + 16 = 112
8	1	1	1	1	1	1	1	1	128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = 255

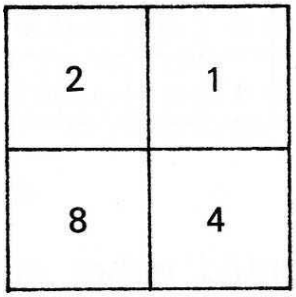
So, for row 1 you could either enter BIN ØlØlØlØl, for example, or enter 85 if you've taken the bother to work it out as a decimal number. Line 7Ø works out where to POKE this number. Line 8Ø keeps printing the new character at the top left-hand corner of the screen so you can see it changing. Line 1ØØ sends the program back to line 1Ø to redefine another character if need be. Note that line 5 allows you to work in the new character set. You may like to add LINE to the INPUT statement in line 1Ø, especially as you have to enter double quotes to redefine ''.

THE BLOCK GRAPHICS

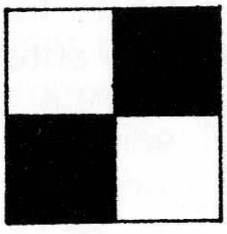
These are the "square" graphics symbols CHR$ 128 to CHR$ 143 which look like this (see also Spectrum manual Appendix A):



There may not appear to be anything special about their orders and CODEs as they appear to be in no particular order. However, you may be interested to know that there is a relationship between their CODE and which of the quarter squares are inked in. In fact, jot down one of these graphics characters and you can work out its CODE from this diagram:



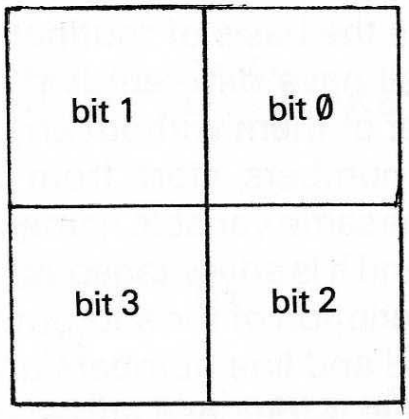
Where each of the quarter squares are inked in, add the numbers shown in the diagram to 128 and the total is the CODE of the character you jotted down. For example, take the symbol:



This would be
CHR$(128 + 1 + 8)
which is CHR$ 137

Another way of looking at this is to consider the individual bits of the CODE of the character. Bits Ø to 3 of the CODE are the significant ones:

If bit Ø is set to 1 the top right quarter of the character is inked in. If bit 1 is set to 1 the top left quarter of the character is inked in. If bit 2 is set to 1 the bottom right quarter of the character is inked in.
If bit 3 is set to 1 the bottom left quarter of the character is inked in.



LIBRARY OF SUBROUTINES

This section is concerned with providing you with a collection of subroutines which you can save on tape individually then MERGE them into your programs later as needed. Of course, they can be used as the basis of routines tailor-made for your application. They all have different line numbers so you can MERGE any number of them without any being overwritten or overlapping. Line numbers start from 9000. Some of the subroutines have the same variable names where the same function is performed and it is advantageous to use the same variable names. The description of the subroutines include details of variable names used and line numbers used, so that you may know what to change if the need arises.

1. Clearing a part of the display

Line numbers used: 9000 to 9045

Variable names used: LINES,F

This routine will clear the number of lines at the bottom of the screen specified by the programmer to the current permanent attributes. The PRINT position is moved to the top left of the area cleared, and resets the PLOT position to the bottom left of the screen (co-ordinates 0,0) as it would after CLS. You are asked to enter a number from 0 to 22 to tell the routine how many lines to erase from the bottom of the display. You may prefer to omit this INPUT line and simply specify the variable LINES before the subroutine is called. If changing the message, take care to keep it short enough not to scroll anything else on the screen. Line 9025 prints a string of 32 spaces on every line to be cleared and provides OVER 0 locally in case OVER 1 is in effect global, in which case nothing would be erased! There is no colour control specified so you may include something like PAPER 8; BRIGHT 8; FLASH 8; locally to preserve the attributes of the background — delete text and graphics but leave the background the same. Line 9010 rejects invalid INPUTs by making you re-enter them. Line 9035 moves the PRINT position to the top left of the part of the screen cleared. Line 9040 moves the PLOT position to the bottom of the screen by POKEing 0 into the system variables that hold the current PLOT co-ordinates.

```
9000 REM  clear part display
9005 INPUT "How many lines? ";LI
NES
9010 IF LINES<0 OR LINES>22 OR L
INES<>INT LINES THEN GO TO 9005
9015 IF LINES=0 THEN RETURN
9020 FOR F=21 TO 22-LINES STEP -
1
9025 PRINT AT F,0; OVER 0;"
                           "
9030 NEXT F
9035 PRINT AT F+1,0;
9040 POKE 23677,0: POKE 23678,0
9045 RETURN
```

2. Yes or No replies

Line numbers used: 9050 to 9080

Variable names used: R$

This routine allows response to the type of questions that need a YES or NO answer. Any word beginning with n or N is turned into an N and any word beginning with y or Y is turned into a Y — from this one alternative course of action can be taken. A yes or no reply must be given — the routine will not allow you to continue unless you enter a word beginning with y,Y,n, or N. On returning from the subroutine, R$ will contain either Y or N for YES or NO respectively. Lower case letters are converted to upper case. Line 9055 fixes the length of R$ (the reply string) at one character. This means that if just ENTER is pressed the reply string is a space which is invalid so must be re-entered. If a word of more than one letter is entered, only the first is put into R$ because of the dimensioning. This ensures that line 9075 need only make sure that R$ is either Y or N.

```
9050 REM yes or no?
9055 DIM R$(1)
9060 INPUT "Yes or no? ";R$
9065 IF R$="y" THEN LET R$="Y"
9070 IF R$="n" THEN LET R$="N"
9075 IF R$<>"Y" AND R$<>"N" THEN
 GO TO 9060
9080 RETURN
```

3. Engage CAPS LOCK

Line numbers used: 9085 to 9100

Variable names used: ZZZ

This program kindly sets CAPS LOCK on for you automatically by means of a tiny six byte machine code routine that can be entered from the keyboard without the need for any form of loader program.

When scanning the keyboard it is usually desirable to check for either upper case or lower case letters only — it is usually inconvenient to have to look for both. The problem is that you can never be absolutely sure if the user is going to engage CAPS LOCK or not. All the routine does is set bit 3 of system variable 23568 to 1 to engage CAPS LOCK. If the program then looks at the keyboard in a way in which the user cannot change the CAPS setting (i.e. use INKEY$ or IN to scan the keyboard) we need not bother checking if the response is upper or lower case. The machine code is contained in a REM statement in line 9Ø95 (which should not contain anything but these six characters after the REM — no colour controls, etc., which would almost certainly confuse the poor machine).

! (exclamation mark)

j (lower case j)

\ (character 92. Note that this is *not* the division symbol. It is the character on the D key obtained in extended mode, SHIFT D)

THEN (the keyword THEN, not the four letters T H E N)

OVER (the statement OVER, not the four letters O V E R)

<> (not equal to symbol, or SYMBOL SHIFT W)

The machine code routine is called with the help of a system variable called NXTLIN, which helps us find where in memory the routine lies as we need this information to know where to start the machine code. NXTLIN actually gives the address of the start of the next program line as its name implies. There are two bytes for line number, two bytes for line length marker and one for the REM character. That's why 5 is added in line 9Ø95. The variable ZZZ is not important to the routine because it only accepts the number which is returned by the machine code routine. It can be *any* variable that is not used.

```
9085 REM engage caps lock
9090 LET ZZZ=USR (PEEK 23637+256
*PEEK 23636+5)
9095 REM !j\ THEN OVER <>
9100 RETURN
```

4. Disengage CAPS LOCK

Line numbers used: 91Ø5 to 912Ø

Variable names used: ZZZ

This does the reverse of the previous routine; it disengages CAPS LOCK so that lower case can be detected. This is, of

course over-ridden, when using INKEY$ simply by pressing the CAPS SHIFT key, so is a bit less useful. Except for the machine code routine and line numbers, the routine is similar to the previous one. The six characters after the REM in line 9115 are:

!	(exclamation mark)
j	(lower case J)
\	(character 92 extended mode, SHIFT D *not* the division symbol)
THEN	(keyword THEN, not the separate 4 letters T H E N)
O	(Graphics O, i.e. CHR$ 158)
<>	(not equal to symbol, SYMBOL SHIFT W)

```
9105 REM disengage CAPS LOCK
9110 LET ZZZ=USR (PEEK 23637+256
*PEEK 23638+5)
9115 REM !j\ THEN O<>
9120 RETURN
```

5. Press any key to continue

Line numbers used: 9125 to 9140

NO variable names used

It is often necessary to make a program wait until a signal is received from the operator, for example, while instructions are read. However, most programmers include the line "Press any key to continue" and their routines do not take pressing the SHIFT keys into account, so you cannot truly press any key in order to continue. This routine scans the bottom row of the keyboard with the function IN to check for the CAPS SHIFT and the SYMBOL SHIFT keys being pressed. The routine may fail if 2 keys are pressed simultaneously on the top three rows of the keyboard, but surprisingly this is very difficult to do.

```
9125 REM Press any key
9130 PRINT "Press any key to con
tinue."
9135 IF INKEY$="" AND IN 65278=2
55 AND IN 32766=255 THEN GO TO 9
135
9140 RETURN
```

6. Using SCREEN$ to detect user defined graphics on the screen

Line numbers used: 9145 to 9195

Variable names used: X,Y,A,B,A$

A shortcoming of the function SCREEN$ is that it can only detect characters with CODEs of 32 to 127 and their inverses on the screen. SCREEN$ works by picking up the address of the start of the character generator from the system variables and looking up in this for a match of the character on screen. To make SCREEN$ detect graphics, it is necessary to make the computer think that the main character set starts at the same place as the user defined graphics so that these dot patterns may be checked for. A little arithmetic manipulation will then get SCREEN$ to return a character with the right CODE. To use the routine you need to specify two variables, Y and X. These are for SCREEN$ (Y,X),Y being the number of the Y co- ordinate down the screen and X being the number of the X co-ordinate across the screen you wish to examine. The original values of the system variables holding the address of the current character generator are preserved in variables A and B so that they may be reinstated after the routine is used. This is done so that you can use a character set other than one in the ROM and the routine will reinstate the correct one. If you have more than one set of user defined graphics the routine will only check against the one currently in use. The routine checks for a normal character first, then if this fails the user defined graphics are checked. If this leads to a character being identified, the CODE of that character is changed to prevent UDG A being confused with SPACE for example. This is done in lin 9180.

```
9145 REM SCREEN
9150 LET A$=SCREEN$ (Y,X)
9155 IF A$<>"" THEN RETURN
9160 LET A=PEEK 23606: LET B=PEE
K 23607
9165 POKE 23606,PEEK 23675
9170 POKE 23607,PEEK 23676-1
9175 LET A$=SCREEN$ (Y,X)
9180 IF A$<>"" THEN LET A$=CHR$
(CODE A$+112)
9185 POKE 23606,A
9190 POKE 23607,B
9195 RETURN
```

7. Search for a copy of a string within another string

Line numbers used: 9200 to 9235

Variable names used: P,B$,C$

This routine allows you to search within one string for a copy of another string. This may be useful in, say, an adventure game where you may wish to detect certain words entered, or in a

filing system where you may wish to search for data. If a copy of C$ is found in B$ then on return from the subroutine, the variable P would contain the number of the element at which the copy starts. For instance, looking for READ in DREADED would return 2. If no copy is found then P is returned with a value of 0. B$ is the main string, and we look in B$ for a copy of C$. The routine will cope with strings of any length within the limits of the memory available. However, very long strings will take a long time to scan.

```
9200 REM INSTR
9205 LET P=0
9210 IF LEN C$=0 OR LEN B$=0 OR
LEN C$>LEN B$ THEN RETURN
9215 FOR P=1 TO LEN B$-LEN C$+1
9220 IF B$(P TO P+LEN C$-1)=C$ T
HEN RETURN
9225 NEXT P
9230 LET P=0
9235 RETURN
```

8. Convert decimal numbers to a binary string

Line numbers used: 9240 to 9280

Variable names used: DEC,D$,J)

When working with memory locations it is sometimes necessary to examine the states of individual bits, or the patterns of ones and zeros of a binary number. Apart from that, this routine can be used for many mathematical applications where it is necessary to convert backwards and forwards between decimal and binary. If you have a number J, then this routine will convert it to an eight or sixteen digit string D$ so that you can imagine each digit as a byte. So the numbers 0 to 255 would emerge as an eight character string. 256 to 65535 would emerge as sixteen character strings. If you do not want this "automatic length" facility just delete lines 9270 and 9275. Before calling the routine define J as the number you wish to convert to binary, (e.g. LET J = 255) then use GOSUB 9240. DEC is made to be a copy of J so that the value can be changed as needed within the routine but preserve the value of J on returning from the subroutine. Repeated division by two helps us build D$ into a string which is the binary equivalent of J. If you wish to convert D$ into a decimal number use VAL ("BIN" + D$).

```
9240 REM DECIMAL TO BINARY
9245 LET DEC=J
9250 LET D$=""
```

```
9255 LET D$=STR$ (DEC-INT (DEC/2
) *2) +D$
9260 LET DEC=INT (DEC/2)
9265 IF DEC<>0 THEN GO TO 9255
9270 IF LEN D$<8 THEN LET D$="00
000000"( TO 8-LEN D$) +D$
9275 IF LEN D$<16 AND LEN D$>8 T
HEN LET D$="00000000"( TO 16-LEN
 D$) +D$
9280 RETURN
```

9. Convert a binary string to a decimal number

Line numbers used: 9285 to 9315

Variable names used: DEC,SUM,E,E$

To use the subroutine define the string E$ as a string of binary digits with 1 representing the binary ones and Ø representing the binary zeros. For example, LET E$ = "11Ø1". GOSUB 9285 will then make DEC the decimal equivalent of E$. Of course, this could be done with binary numbers using BIN, but it is occasionally necessary to store the information as a string so that the individual bits may be checked. These routines allow you to work in binary and decimal numbers and convert backwards and forwards as you need to. It is possible to use BIN to convert a binary string to a decimal number (in a very roundabout way using VAL) as shown in the second alternative, but this is subject to the constraints of VAL and BIN.

```
9285 REM BINARY TO DECIMAL
9290 LET DEC=0: LET SUM=1
9295 FOR E=LEN E$ TO 1 STEP -1
9300 IF E$(E)="1" THEN LET DEC=D
EC+SUM
9305 IF E$(E)="0" OR E$(E)="1" T
HEN LET SUM=SUM*2
9310 NEXT E
9315 RETURN
```

```
9285 REM BINARY TO DECIMAL
9290 LET DEC=VAL ("BIN "+E$)
9295 RETURN
```

With both versions of the programs you can include spaces between the digits of E$ to aid reading by clearly spacing out the numbers like this: the string "1111111111111111" (sixteen ones) could be entered as "1111 1111 1111 1111" without affecting the values because the routine just skips over any characters that are not ones or zeros. Using BIN you could separate the digits of the binary number with spaces but any other character would cause an error C, Nonsense In Basic.

10. Round off to 2 decimal places

Line numbers used: 9320 to 9345

Variable names used: F$,VALUE,LF

This routine rounds off the variable VALUE to 2 decimal places and adds a zero before the decimal place if required. This makes it very useful for calculations involving money. The value is returned as the string F$ ready for printing or decoding with VAL if it is required to be handled as a number. For example, if value was 0·678 the string variable F$ would be 0·68. You should add the £ or $ symbol as you require during printing. Before calling the subroutine declare VALUE along the lines of LET VALUE = 89·2345 or INPUT "Enter the cost of the item"; VALUE. Due to the way the Spectrum holds decimal numbers in memory, a slight inaccuracy can sometimes occur where the third decimal place is 5. 0·005 will be treated incorrectly, ending up as 0 instead of 0·01 — there is a simple answer, namely adding slightly more than 0·5 in line 9325 — something like 0·500001.

```
9320 REM 2 DEC. PLACES
9325 LET F$=STR$ (INT (VALUE*100
+.5)/100)
9330 IF F$(1)="." THEN LET F$="0
"+F$
9335 LET LF=LEN F$-LEN STR$ INT
VAL F$
9340 LET F$=F$+(".00" AND LF=0)+
("0" AND LF=2)
9345 RETURN
```

11. Keyboard aids for graphics games

Line numbers used: 9350 to 9380

Variable names used: X,Y

The routines described here show how whole sections of the keyboard may be used to control screen movement in particular directions. No more frantic searching for the 5 and 8 keys, for instance.

In the first routine, the left half of the keyboard moves towards the left (not literally anyway) and the right half of the keyboard does the same to the right. With a choice of 20 keys to move you in the direction, you shouldn't have any more trouble finding the right keys for movement. The only thing to watch

out for is not to press CAPS SHIFT and SPACE causing BREAK and stopping the program.

The control effects a change of value of the variable X — this could be used as the PRINT or PLOT co-ordinate of an object on the screen. It is up to you to determine the minimum and maximum values which X can take.

```
9350 REM MOVE LEFT OR RIGHT
9355 LET X=X+(IN 61438+IN 57342+
IN 49150+IN 32766<>1020)-(IN 634
86+IN 64510+IN 65022+IN 65278<>1
020)
9360 RETURN
```

The second routine provides control for movement in eight directions by splitting the keyboard into four sections — the top row for up, the left half of the middle two rows for left, the right half of the middle two rows for right and the bottom row of keys for down. Pressing suitable combinations of these can move you diagonally. X is the co-ordinate of the PRINT position across the screen and Y is the co-ordinate of the PRINT position down the screen. Again you should set the limits of the values these variables can take yourself.

```
9365 REM 4 DIRECTION MOVE
9370 LET X=X+(IN 49150+IN 57342<
>510)-(IN 64510+IN 65022<>510)
9375 LET Y=Y+(IN 65278+IN 32766<
>510)-(IN 63486+IN 61438<>510)
9380 RETURN
```

12. String sorting

Line numbers used: 9385 to 9415

Variable names used: S,T,S$,U$,USED

This routine will allow you to sort a string array into alphabetical order provided you know how many of them there are to be sorted. USED is a variable that says how many strings there are in the string array. So if you had earlier declared DIM S$ (20,20) and had assigned to 15 strings, USED should have a value of 15 because there's no point in sorting the other 5. The other two varvables usea are two loop control variables and a dummy string used during the swapping over of two strings.

```
9385 REM sort strings
9390 FOR S=1 TO USED-1
```

```
9395 FOR T=S TO USED
9400 IF S$(T)<S$(S) THEN LET U$=
S$(S): LET S$(S)=S$(T): LET S$(T
)=U$
9405 NEXT T
9410 NEXT S
9415 RETURN
```

13. Printing string arrays without any trailing spaces

Line numbers used: 9420 to 9440

Variable names used: U,W,S$

This routine allows you to print strings or string arrays without the problem of spaces at the end messing up the screen layout by causing a gap between two words. To use the routine replace string array with the name of the string array you wish to print. W indicates which string in the array is to be printed. So if you had DIM S$ (20,20) and wished to print S$ (8) you would say LET W = 8 before calling the subroutine.

```
9420 REM PRINT LESS END SPACES
9425 FOR U=LEN S$(W) TO 1 STEP -
1
9430 IF S$(W,U)<>" " THEN PRINT
S$(W, TO U): RETURN
9435 NEXT U
9440 RETURN
```

14. Make a bleep

Line numbers used: 9445 to 9465

Variable names used: Z

The routine merely provides a tone of descending pitch which is suitable for any application requiring a distinct sound — say, where you have just been shot down by an alien.

```
9445 REM MAKE A NOISE
9450 FOR Z=0 TO 20 STEP 2
9455 BEEP .01,69-(Z*2.5)
9460 NEXT Z
9465 RETURN
```

15. Decimal to hexadecimal conversion

Line numbers used: 9470 to 9505

Variable names used: DEC,NUMBER,H$,N1

Suppose you had a decimal number DEC which you wanted to convert to hexadecimal format for machine code programming or whatever reason. This subroutine will convert the value of

DEC into a hex string H$ so that 16 decimal would be 10 hex. There is no limit to the size of the number DEC (within the arithmetic limits of the Spectrum) as long as it is positive. Since two hex digits normally equate with one byte in memory, this routine adds a leading zero digit so that the resultant hex string H$ can be used with hex loader programs if needed. This function is performed by line 9500 so you may omit this if you do not need the facility. The other variables are dummies used to perform arithmetic on the value of DEC during conversion, without actually changing the value of DEC — it is preserved on exit from the routine.

```
9470 REM decimal to hex
9475 LET dec=number
9480 LET h$=""
9485 LET n1=INT (dec-(INT (dec/1
6))*16)
9490 LET h$=CHR$ (n1+48+(7 AND n
1>9))+h$
9495 IF INT (dec/16)<>0 THEN LET
 dec=INT (dec/16): GO TO 9485
9500 IF INT (LEN h$/2)*2<>LEN h$
 THEN LET h$="0"+h$
9505 RETURN
```

16. Hexadecimal to decimal conversion

Line numbers used: 9510 to 9540

Variable names used: H$,H,VALUE,DEC

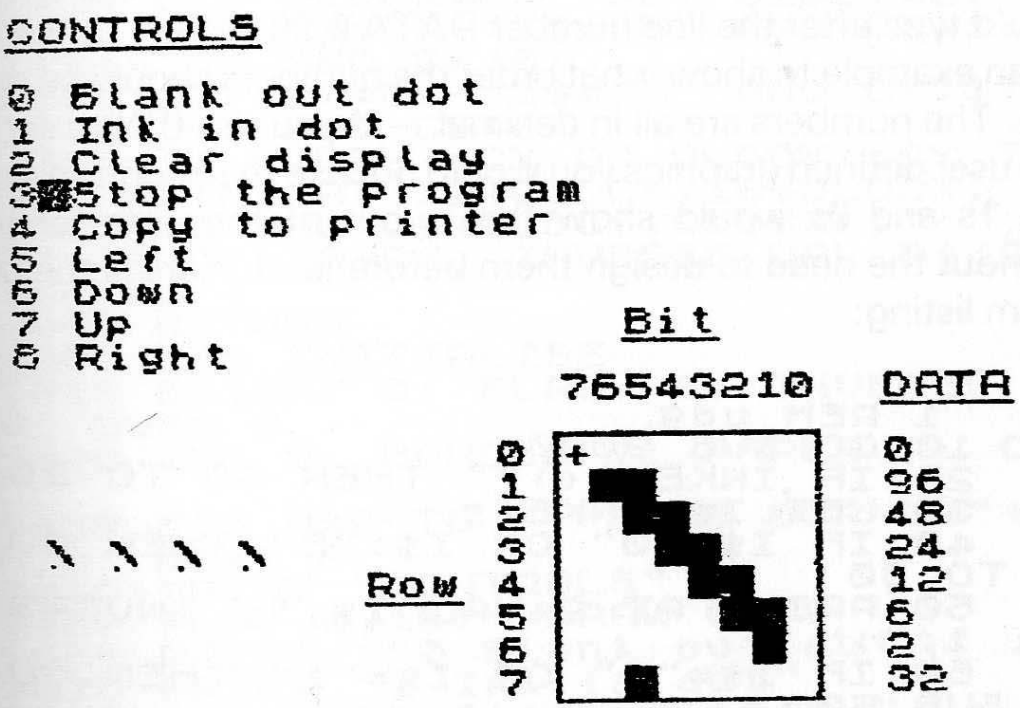
This routine performs the opposite function, namely converts a hexadecimal string H$ to a decimal number DEC. H$ may be any length as long as the resultant decimal number is within the arithmetic capabilities of the Spectrum. H$ should be defined before the subroutine is called, it is preserved on exit from the subroutine and the decimal number is in the variable DEC.

The characters of the hex string should consist of the numbers 0 to 9, the upper case letters A to F or the lower case letters a to f (which are read as being the same whether in lower or upper case).

```
9510 REM hex to decimal
9515 LET value=1: LET dec=0
9520 FOR h=LEN h$ TO 1 STEP -1
9525 LET dec=dec+(CODE h$(h)-48-
(7 AND h$(h)>"9")-(32 AND h$(h)>
="a"))*value
9530 LET value=value*16
9535 NEXT h
9540 RETURN
```

UDG DESIGNER (16K SPECTRUM)

This program will allow you to design the user defined graphic of your choice and then tell you the values to use as data in a program to set up those graphics. It does not include the many luxuries of some more elaborate versions, but having used one of those I find that when I need to design a character for a program this simple version is all I need. All the commands and options available are shown on the screen all the time and if you have a printer you have the option of having a permanent copy of the display including a large size image, what the character looks like normal size and the data values. This is a sample of the display so that you can have an idea of what the program's like.



The large square is the display area where the character you're designing is displayed. Study this area when using the program. The cursor (a + symbol) shows the current dot of the graphic that can be shaded in or blanked out. This cursor can be moved left, down, up or right with the cursor keys 5,6,7 and 8 respectively. Pressing the 0 key will blank out the dot under the cursor so if the cursor + was in the top left of the box then the top left dot would be blanked out white after pressing 0 no matter whether it was black or white previously. On the other hand, pressing the 1 key would make the dot black. Note that

the + cursor is white against the black dots so that you can always see it. Four "real" size copies of the user defined graphic are printed side by side on the left of the enlarged version so that you can see what it should actually look like in use.

Pressing the 2 key clears the display of the user defined graphic, resets the cursor to the top left of the box and puts the data values back to Ø each. Pressing the 3 key will stop the program if you've finished with it. If for any reason you want to restart the program from where you stopped it (e.g. in error) then simply entering CONTINUE as a command will step over the STOP statement. Pressing the 4 key will give you a printout on paper (if, of course, the ZX printer is attached), without the cursor. Once you've finished designing, copy the data values so that you can incorporate them in a DATA statement for the user defined graphics in your program. For the example shown you'd type after the line number DATA Ø,96,48,24,12,6,2,32 as an example to show what order the numbers should be written. The numbers are all in decimal — if you use BIN to set up the user defined graphics you would not use this program since the 1s and Øs would show the layout of the dots for you without the need to design them beforehand. Here is the program listing:

```
  1 REM udg
 10 GO SUB 9000
 20 IF INKEY$<>"" THEN GO TO 20
 30 LET I$=INKEY$
 40 IF I$<"0" OR I$>"8" THEN GO
TO 30
 50 PRINT AT 2+VAL I$,1; INVERS
E 1;">"
 60 IF I$="0" OR I$="1" THEN GO
SUB 500
 70 IF I$="2" THEN GO SUB 1000
 80 IF I$="3" THEN STOP
 90 IF I$="4" THEN GO SUB 1500
100 IF I$>="5" AND I$<="8" THEN
GO SUB 2000
110 PRINT AT 2+VAL I$,1;" "
120 GO TO 20
500 REM ink in or blank out
510 LET D$(CY,CX)=I$
520 LET D(CY)=VAL ("BIN "+D$(CY
))
530 POKE USR "A"-1+CY,D(CY)
540 PRINT AT 12+CY,16+CX; INVER
SE VAL D$(CY,CX);"+"; INVERSE 0;
AT 16,1;"A A A A";AT 12+CY,27;D(
CY);"   "( TO 3-LEN STR$ D(CY))
```

```
 550 RETURN
1000 REM clear the display
1010 FOR A=0 TO 7
1020 POKE USR "A"+A,0
1030 LET D$(A+1)="00000000"
1040 LET D(A+1)=0
1050 PRINT AT 13+A,17;"          "
;AT 16,1;"A A A A";AT 13+A,27;"0
"
1060 NEXT A
1070 LET CY=1: LET CX=1
1080 PRINT AT 12+CY,16+CX;"+"
1090 RETURN
1500 REM copy to printer
1510 PRINT AT 12+CY,16+CX; OVER
1;" +"
1520 COPY

1530 PRINT AT 12+CY,16+CX; OVER
1;" +"
1540 RETURN
2000 REM move cursor
2010 LET OLDCX=CX: LET OLDCY=CY
2020 LET CX=CX-(I$="5" AND CX>1)
+(I$="8" AND CX<8)
2030 LET CY=CY-(I$="7" AND CY>1)
+(I$="6" AND CY<8)
2040 IF OLDCX<>CX OR OLDCY<>CY T
HEN PRINT AT 12+OLDCY,16+OLDCX;
INVERSE VAL D$(OLDCY,OLDCX);" ";
AT 12+CY,16+CX; INVERSE VAL D$(C
Y,CX);"+"
2050 RETURN
9000 REM INITIALISE
9010 BRIGHT 0: FLASH 0: INVERSE
0: OVER 0
9020 INK 0: BORDER 7: PAPER 7: C
LS
9030 FOR A=0 TO 7: POKE USR "A"+
A,0: NEXT A
9040 PRINT "CONTROLS"
9050 PLOT 0,167: DRAW 63,0
9060 PRINT ;"0 Blank out dot"
9070 PRINT "1 Ink in dot"
9080 PRINT "2 Clear display"
9090 PRINT "3 Stop the program"
9100 PRINT "4 Copy to printer"
9110 PRINT "5 Left"
9120 PRINT "6 Down"
9130 PRINT "7 Up";TAB 19;"Bit":
PLOT 152,95: DRAW 23,0
9140 PRINT "8 Right"
9150 PRINT TAB 17;"76543210  DAT
A": PLOT 216,79: DRAW 31,0
9160 FOR A=0 TO 7: PRINT TAB 15;
;A;TAB 27;"0": NEXT A
9170 PLOT 135,72: DRAW 65,0: DRA
W 0,-65: DRAW -65,0: DRAW 0,64
9180 PRINT AT 17,11;"Row": PLOT
88,31: DRAW 23,0
```

```
9190 PRINT AT 13,17;"+"
9200 DIM D(8): DIM D$(8,8)
9210 FOR A=1 TO 8: LET D$(A)="00
000000": NEXT A
9220 LET CY=1: LET CX=1
9230 RETURN
```

When typing in the listing, please bear the following information in mind.

Line 520: The word BIN in quotes after VAL is the function BIN, not the three letters B, I and N.

Line 530: In quotes after USR, the A is a graphics A.

Line 540: The letter As in "A A A A" are four graphics As.

Line 1050: There are 8 spaces in the first string of spaces and 4 graphics As in "A A A A" and "0 " is followed by two spaces.

Line 2040: One space.

Line 9150: Two spaces between the numbers and DATA. Note the two apostrophes after the quotes.

The program is designed as a series of subroutines called from an initial loop in the program. The first thing done is to initialise the program in the subroutine at line 9000. This consists of setting permanent attributes, etc. (9010 to 9020, black text on white background), making the user defined graphic on key A have all dots blanked out to match the blank box at the start. This is done by the standard POKE USR statement 8 times in a loop.

Lines 9040 to 9140 print the instructions. The DRAW statements are for underlining. Lines 9150 to 9180 set up the box in which you edit your character. The numbers along the top indicate which bit of the DATA byte the dot corresponds to. The numbers along the side indicate which byte of the data for the user defined graphic, e.g. row 0 will go into USR "A" + 0, etc.

Line 9190 prints the cursor at its starting position. Line 9200 sets up one numeric array D() and a string array D$(). The former holds the DATA values whereas the string array holds a binary version to determine whether the individual dots are set or not. If you want a printout in binary for BIN users this array

could be printed. Line 9210 puts 0 into every element of D$() since the display starts with all dots "off". The variables in line 9220 are CY for cursor Y co-ordinate down the screen (note, *not* from the top of the screen) and CX for cursor X across, again not standard X values. On returning from the subroutine the program enters the main loop in lines 20 to 120. Line 50 prints an inverse greater-than symbol as a cursor next to the option chosen. This is optional and can be omitted (in which case omit also line 110 which deletes the marker once the option has been completed which in most cases takes only a fraction of a second). Lines 60 to 100 select which subroutine to execute. If adding any routines of your own or changing this program be careful as direct references to the character in I$ are made in other parts of the program; in particular, VAL is applied and the value used to determine whether something is in inverse or not.

The subroutine at line 500 is called when you press the 0 or 1 key. It handles inking in or blanking out a dot. Firstly, the appropriate element of D$() is made 0 for somewhere blanked out or made 1 for somewhere inked in. This is converted to decimal in line 520 by applying VAL to a string containing BIN and a binary string from D$() — the result is stored in the appropriate element of the array D() with the data. D() is updated every time any change is made to the UDG. This value is POKEd into the user defined graphics area in memory so that graphics A becomes a copy of the figure you're editing. Line 540 prints the + cursor in the display box and determines whether this is to be inverse or not from the character in I$ you pressed on the keyboard. It also updates the data values on the screen by printing the appropriate element of D() in the column to the right of the display box. Note the use of LEN STR$ to decide how many spaces to print to overwrite any value that may have been there previously — imagine that the DATA for a row was 127 (i.e. only bit 7 was blanked out) then you blanked out bit 6, which made the data 63. This would only have 2 digits instead of the three there previously so would not completely overwrite it and leave the 7 on the screen giving 637 — oops! The last PRINT statement in line 540 decides how many spaces to print, 2,1, or none depending on how many characters in the number. Three characters are always printed. Line 540 also prints the 4 graphics As.

The next subroutine at line 1000 clears the display of the graphics you have set up and resets the data arrays to starting values as well as the cursor and UDG A.

The routine at line 1500 copies the display to the ZX printer. The + cursor is first erased using PRINT OVER 1 which, since the cursor is already inverse against the background, will turn the cursor the same as the background, so in effect it is erased. COPY is used when the cursor is returned using PRINT OVER 1 again. This is a very useful application of OVER 1, erasing and reprinting something on the screen alternately simply by printing the same thing OVER and OVER again. The subroutine at line 2000 is a straightforward routine to move the cursor using the cursor keys 5 to 8. Note in line 2040 how the binary strings D$ is used again with VAL to determine whether or not something is to be printed in inverse. Unusual, but ideal for this application.

You may notice there is no repeat on the keys, e.g. if you want to move from one side of the box to the other you have to press the 8 key once for every space the cursor moves. This is due to line 20 which if it finds a key is pressed waits for it to be released. If you don't see the point of this try omitting it. Because the program works so fast the keyboard becomes very difficult to use because the cursor travels so fast. If you like, experiment with using a fixed delay in line 20, something like 20 FOR A = 1 TO 50: NEXT a which will give a nice slow constant repeat. Do not use PAUSE as this will be cut short after a keypress and you will have to play around with keyboard auto repeat periods and so on.

READY MADE USER DEFINED GRAPHICS

These are pre-designed graphic symbols for you to use in your own graphics programs so that you don't have to waste time designing your own. They consist of the more commonly used user defined graphics, with the relevant data for their creation placed in DATA statements for subsequent reading and POKEing into the user defined graphics area of memory. Although the line numbering is pretty random since you will change this to suit your programs, the DATA lists are shown as they would appear on the screen in a program listing as a visual aid to correct entry.

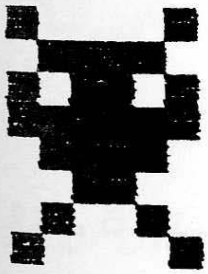
For example, to set up the invader character:

```
8000 REM make  graphic A
8010 REM into an invader
8020 RESTORE
8030 FOR a=USR "a" TO USR "a"+7
8040 READ udg
8050 POKE a,udg
8060 NEXT a
8070 REM data list taken from
8080 REM examples and renumbered
8090 REM to suit program
8100 DATA 66,60,90,126,60,24,36,
66
8110 PRINT "A": REM graphic A
```

Of course, you would leave out the REM statements for simplification. And that's all there is to it — go ahead and use them!

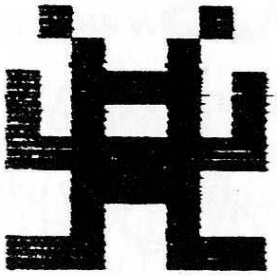
(1) invader characters

```
invader1
```



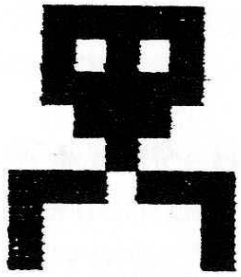
```
8010 DATA 66,60,90,126,60,24,36,
66
```

invader2



```
8030 DATA 66,36,189,165,255,60,3
6,231
```

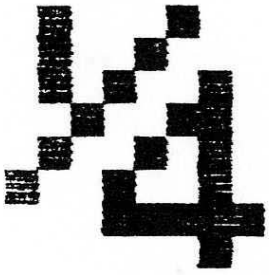
invader3



```
8050 DATA 62,42,62,28,8,119,65,6
5
```

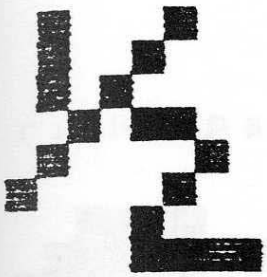
(2) fractions

quarter



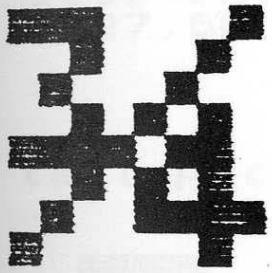
```
8290 DATA 68,72,82,38,74,146,31,
2
```

half



```
8280 DATA 68,72,80,44,66,132,8,1
5
```

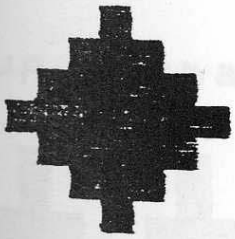
three quarters



```
8310 DATA 225,34,68,42,246,42,79
,130
```

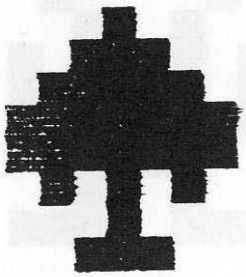
(3) cards

cards diamond



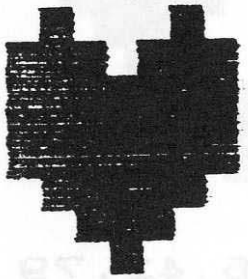
```
8150 DATA 8,28,62,127,62,28,8,0
```

cards spades



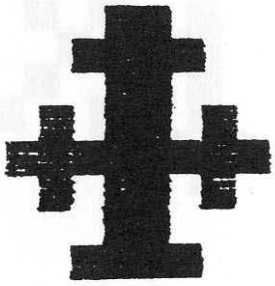
```
8170 DATA 16,56,124,254,254,84,1
6,56
```

cards hearts



```
8190 DATA 68,238,254,254,254,124
,56,16
```

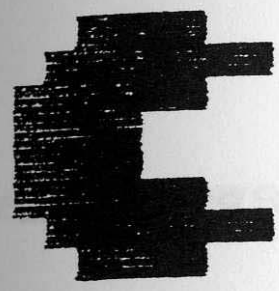
cards clubs



```
8210 DATA 24,60,24,90,255,90,24,
60
```

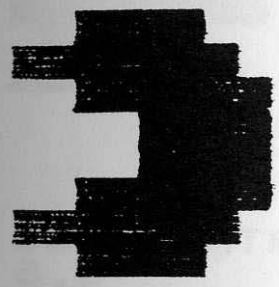
(4) pacman-type symbols

right pacman



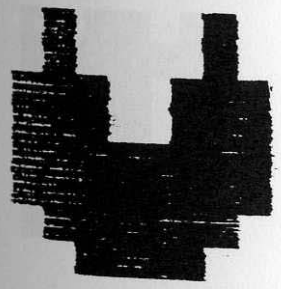
```
8010 DATA 60,127,252,240,240,252
,127,60
```

left pacman



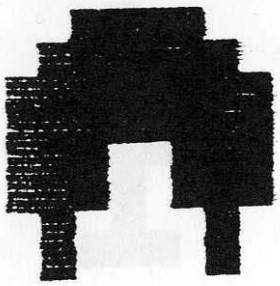
```
8050 DATA 60,254,63,15,15,63,254
,60
```

UP pacman



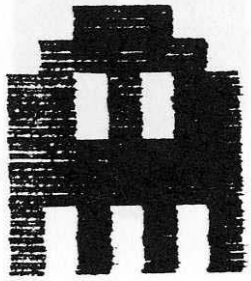
```
8030 DATA 66,66,231,231,255,255,
126,60
```

down pacman



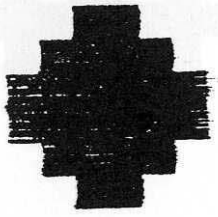
```
8070 DATA 60,126,255,255,231,231
,66,66
```

ghost



```
8090 DATA 56,124,214,214,254,254
,170,170
```

energy tablet



```
8110 DATA 0,24,60,126,126,60,24,
0
```

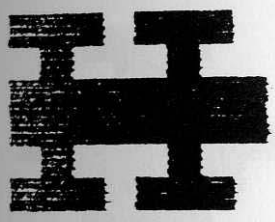
food dot



```
8130 DATA 0,0,0,24,24,0,0,0
```

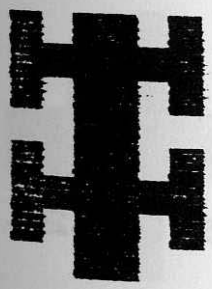
(5) cars

right car



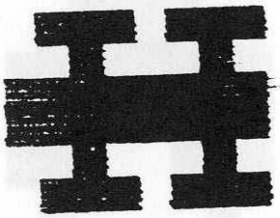
```
8010 DATA 0,238,68,255,255,68,23
8,0
```

down car



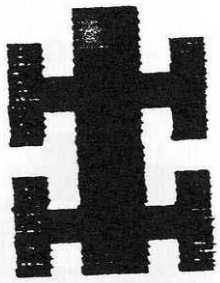
```
8070 DATA 90,126,90,24,90,126,90
,24
```

left car



```
8030 DATA 0,119,34,255,255,34,11
9,0
```

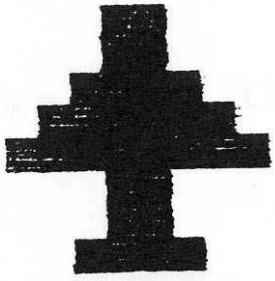
up car



```
8050 DATA 24,90,126,90,24,90,126
,90
```

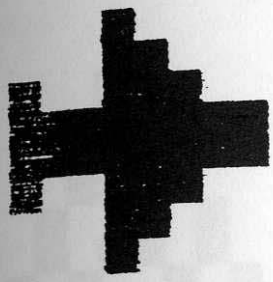
(6) plane

plane1



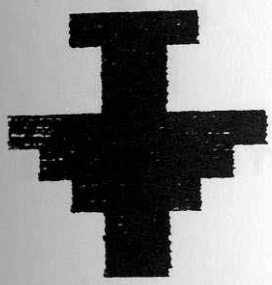
```
8000 DATA 24,24,60,126,255,24,24
,60
```

plane2



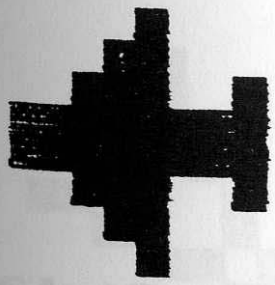
```
8030 DATA 16,24,156,255,255,156,
24,16
```

plane3



```
8050 DATA 60,24,24,255,126,60,24
,24
```

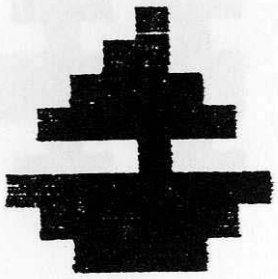
plane4



```
8070 DATA 8,24,57,255,255,57,24,
8
```

(7) ship

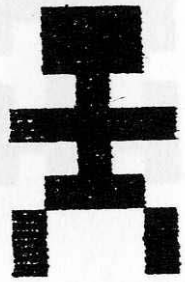
```
ship
```



```
8090 DATA 8,24,60,126,8,255,126,
60
```

(8) man

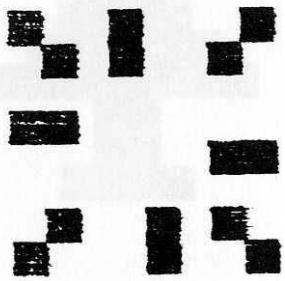
```
man
```



```
8010 DATA 56,56,16,124,16,56,68,
68
```

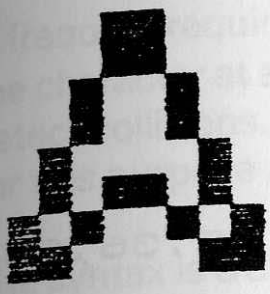
(9) explosion

```
explosion
```

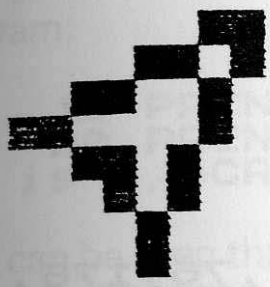


```
8330 DATA 145,82,0,192,3,0,74,13
7
```

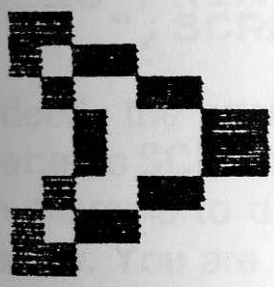
(10) general purpose eight direction craft for space shoot-out games, etc.



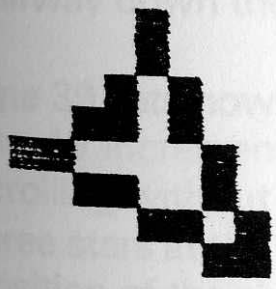
```
8120 DATA 24,24,36,36,66,90,165,
195
```



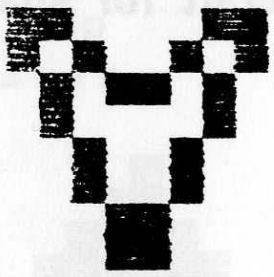
```
8140 DATA 3,13,50,194,52,20,8,8
```



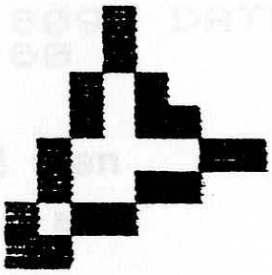
```
8160 DATA 192,176,76,35,35,76,17
6,192
```



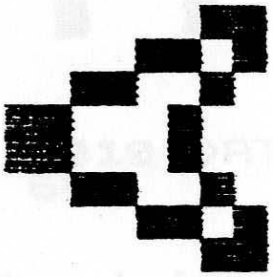
```
8180 DATA 8,8,20,52,194,50,13,3
```



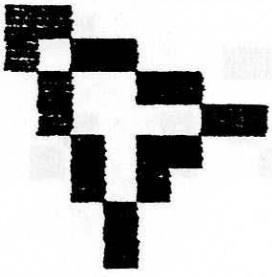
```
8200 DATA 195,165,90,66,36,36,24
,24
```



```
8220 DATA 16,16,40,44,67,76,176,
192
```



```
8240 DATA 3,13,50,196,196,50,13,
3
```



```
8260 DATA 192,176,76,67,44,40,16
,16
```

SORTING OUT SCREEN$ AND ATTR

A frequent requirement during games programming is to check the character at a given location on the screen, for example, to detect collisions. Spectrum BASIC has the SCREEN$ function for this purpose and it works like this.

The syntax is SCREEN$(y,x) which returns a string corresponding to the character on the screen at row y down the screen and column x across the screen irrespective of whether the character is inverse or not. y and x can be numbers or numeric expressions as in a PRINT statement. For instance, try this program:

```
10 PRINT AT 0,5;"+"
20 PRINT "The character at 0,5
is ";SCREEN$ (0,5)
```

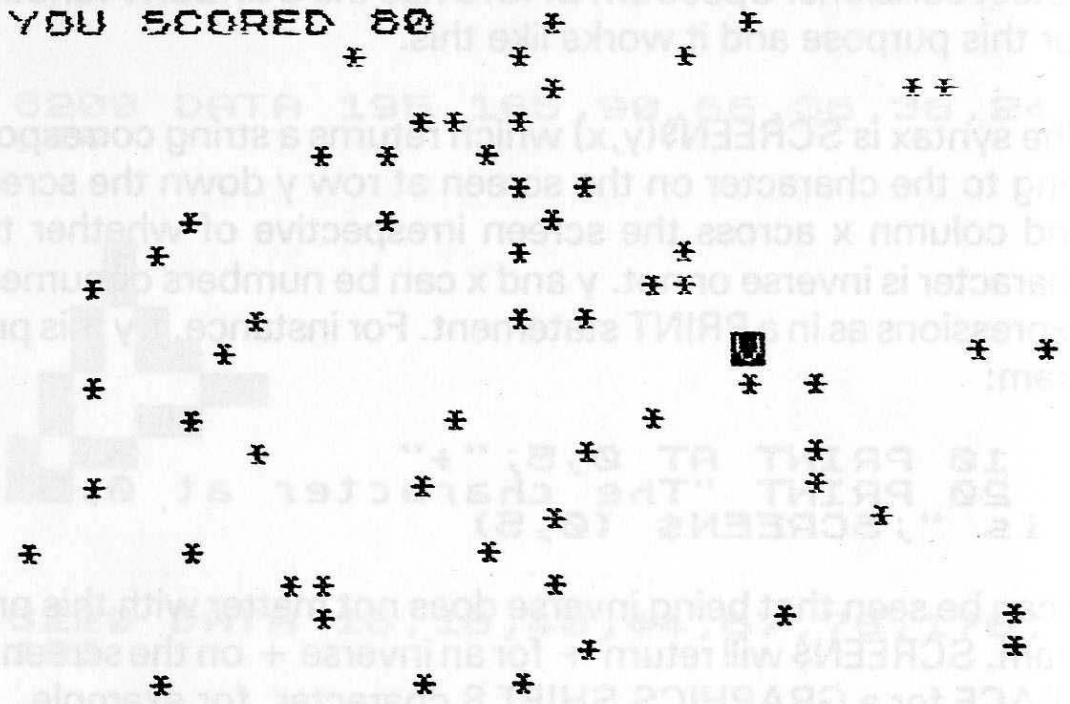
It can be seen that being inverse does not matter with this program. SCREEN$ will return + for an inverse + on the screen or SPACE for a GRAPHICS SHIFT 8 character, for example.

```
10 PRINT AT 0,5; INVERSE 1;"+"
20 PRINT "The character at 0,5
is ";SCREEN$ (0,5)
```

Indeed, the attributes (colours/flashing/brightness) do not matter to SCREEN$, only the actual dots on the screen. Here is an example to demonstrate the use of SCREEN$ in graphic games. You are in control of a spaceship (inverse V) drifting through space trying to avoid colliding with the stars (asterisks) drifting past you. When you collide, the game stops and your score is displayed. The 5 key moves you left across the screen and the 8 key moves you right across the screen. You stay halfway down the screen while the action rushes up past you.

Line 30 sets how far across the screen your spaceship starts, line 40 increments your score and line 50 keeps the screen scrolling without the scroll? prompt appearing. Line 60 prints three stars at the bottom of the screen then erases the current position of the ship for scrolling. The scrolling is performed by the last PRINT statement in line 60. Line 70 looks at the keyboard and determines where to PRINT the spaceship. Line 80

looks for the character at the new position using SCREEN$. If this character isn't an asterisk then the game continues by printing the spaceship and looping back to line 40. If there has been a collision your score is printed and the game ends with your spaceship flashing to indicate a collision.




```
  1 REM stars
 10 RANDOMIZE
 20 LET SCORE=0
 30 LET ACROSS=INT (RND*32)
 40 LET SCORE=SCORE+1
 50 POKE 23692,255: REM SCROLL
 60 PRINT AT 21,RND*31;"*";AT 2
1,RND*31;"*";AT 21,RND*31;"*";AT
 10,ACROSS;" ";AT 21,31;
 70 LET ACROSS=ACROSS-(INKEY$="
5" AND ACROSS>0)+(INKEY$="8" AND
 ACROSS<31)
 80 IF SCREEN$ (10,ACROSS)<>"*"
 THEN PRINT AT 10,ACROSS; INVERS
E 1;"V": GO TO 40
 90 PRINT AT 0,0;"YOU SCORED ";
SCORE;AT 10,ACROSS; FLASH 1;"V"
```

What you don't get told is that SCREEN$ is very limited in what it can recognise. It only checks against those characters in the character set from SPACE to the © symbol, or CHR$(32) to CHR$(127). User defined graphics and the block graphics on the number keys on the keyboard are not recognised, which is a great pity since these are the graphics mainly used in graphics based programs where SCREEN$ is of most use.

One way around this problem is to use ATTR, and PRINT in different colours. So if you had a game where you fired missiles at an object you could arrange that the object was green, the missile red and the missile launcher blue and so on. To check if the missile had hit the object you would check for the presence of green on the character square the missile was about to move to, with ATTR. This sort of thing is usually done where SCREEN$ cannot be used for any reason. For instance, if the object was blue on a yellow background of normal brightness and not flashing you would test for the presence of the object with something like:

```
IF ATTR(Y,X)=49 THEN....
```

bearing in mind that ATTR returns a number which is (flashing * 128) + (brightness * 64) + (PAPER colour * 8) + (INK colour).

One thing to watch out for here is that any blank spaces on the screen should have a different attribute to any other graphics on the screen. If the global colour was blue INK and the missiles and the spaces for blanking were both printed in the global colours then ATTR may not be able to distinguish between the spaces and the missiles! One easy way around this is to specify a global INK colour the same as the background PAPER colour before clearing the screen like this:

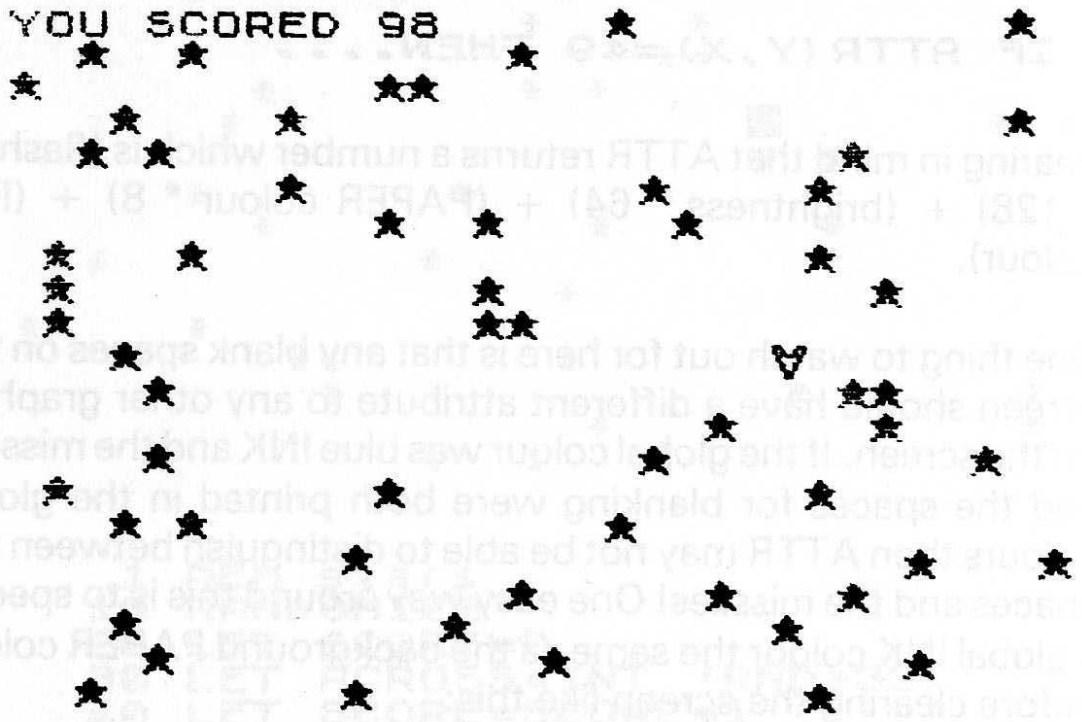
```
FLASH 0: BRIGHT 0: INK 6: BORDER 6: PAPER 6: CLS.
```

This sets the attributes of any blank spots on the screen to 54 (the screen will appear yellow). You will not be likely to use yellow INK on yellow PAPER during the course of the program for printing graphics because it would be rather difficult to see. Anything printed on the screen would be printed with temporary colours while all blanking would be done in the permanent colours. The only problem is that when the program stops you won't be able to see the listing unless you specify something like INK 9: STOP when the program comes to an end.

Here is a development of the stars program which makes use of the user defined graphics for an improved display and ATTR for checking the display for collisions as described above. The

graphics are three graphic A's in line 70 and a graphic B in lines 100 and 110. Line 20 sets the global attributes to 0 so that an area of screen with nothing printed on has an attribute of 0. The stars are printed in white against black spaces to give them an attribute of something other than 0. Thus, when the program checks to see if a collision has occurred in line 100 only empty space allows the action to continue. If there isn't empty space, a collision has occurred. Here is the program listing.

Stars 2 screen display and listing




```
  5 GO SUB 1000
 10 RANDOMIZE
 20 FLASH 0: BRIGHT 0: INK 0: B
ORDER 0: PAPER 0: CLS
 30 LET SCORE=0
 40 LET ACROSS=INT (RND*32)
 50 LET SCORE=SCORE+1
 60 POKE 23692,255: REM AUTO-SC
ROLL
 70 PRINT INK 7;AT 21,RND*31;"*
";AT 21,RND*31;"*";AT 21,RND*31;
"*"
 80 PRINT AT 10,ACROSS;" ";AT 2
1,31;
 90 LET ACROSS=ACROSS-(INKEY$="
5" AND ACROSS>0)+(INKEY$="8" AND
 ACROSS<31)
 100 IF ATTR (10,ACROSS)=0 THEN
PRINT AT 10,ACROSS; INK 4;"V": G
O TO 50
 110 PRINT AT 0,0; INK 7;"YOU SC
```

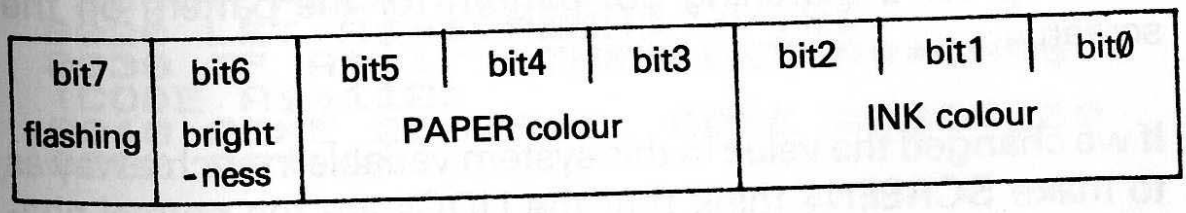
```
ORED ";SCORE;AT 10,ACROSS; INK 4
; FLASH 1;"V"
 120 INK 9: STOP
1000 REM MAKE GR. A INTO STAR
1010 FOR A=0 TO 15
1020 READ UDG
1030 POKE USR "A"+A,UDG
1040 NEXT A
1050 DATA 16,56,254,124,56,108,1
30,0,195,165,90,66,36,36,24,24
1060 RETURN
```

Sometimes it is possible to use either SCREEN$ or ATTR and it becomes necessary to choose which one can be used. In situations where either can be used, speed is usually the determining factor. Try both of these programs and time how long they take to run.

```
10 FOR A=1 TO 1000
20 LET B$=SCREEN$ (2,2)
30 NEXT A
```

```
10 FOR A=1 TO 1000
20 LET B=ATTR (2,2)
30 NEXT A
```

The one using SCREEN$ takes about 12 seconds whereas the version with ATTR takes about 9 seconds. So it can be seen that it is quicker to return the attributes of a screen location than to return the character at that location. Sometimes you can run into the difficulty of having to find the PAPER colour or the INK colour or whether the location is bright or flashing. ATTR returns the attributes of a character location as one number from 0 to 255 which consists of the information for the four of them. To resolve them into their constituent parts we need to know how the attributes are held in memory in the display file. This diagram represents an attribute byte in the display.



bits 0 to 2 hold the INK colour in binary

bits 2 to 5 hold the PAPER colour in binary

bit 6 holds the BRIGHTness attribute, being 1 if bright (or BRIGHT 1)

bit 7 holds the FLASHing attribute, being 1 if flashing (or BRIGHT 1)

To resolve into individual attributes requires some thought. These FN calls will resolve the flashing attribute (FN f), the BRIGHT attribute (FN b), the PAPER attribute (FN p) and the INK attribute (FN i). If you are only going to use the expression once in a program there is no need to set up the 4 FN calls — just write out the expression in full when it's needed. The numbers returned are the colour numbers, so if the PAPER colour was blue, the function FN p would yield 1. The functions' arguments x and y are the standard x and y screen co-ordinates, so to find the PAPER colour of the top left of the screen you would say LET paper = FN p(0,0). x is the column number across the screen and y is the row number down the screen.

```
10 DEF FN f(y,x)=INT (ATTR (y,
x)/128)
20 DEF FN b(y,x)=INT ((ATTR (y
,x)-INT (ATTR (y,x)/128)*128)/64
)
30 DEF FN p(y,x)=INT ((ATTR (y
,x)-INT (ATTR (y,x)/64)*64)/8)
40 DEF FN i(y,x)=INT (ATTR (y,
x)-INT (ATTR (y,x)/8)*8)
```

Inevitably you will find situations where you need to find user defined graphics on the screen and ATTR cannot be used. It is actually possible to make SCREEN$ recognise user defined graphics once you understand how SCREEN$ works. The function SCREEN$ picks up a value from a certain system variable which enables it to find the start of the character set which it looks up for a matching dot pattern for the pattern on the screen.

If we changed the value in this system variable in such a way as to make SCREEN$ think that the UDGs are the normal character set then use a simple arithmetic manipulation to get the correct CHR$ value (it might think that UDG A was CHR$ 32 otherwise, since CHR$ 32 is the first character in the character set normally).

The system variable in question is 23606/23607 which is a two byte system variable containing a number which is 256 less than the address of the start of the character set dot patterns. There is another two byte system variable 23675/23676 that tells us where the user defined graphics start (exactly). So, all we need to do is transfer the two bytes from 23675/6 into 23606/7 subtracting one from the high byte for the difference of 256.

This is best written as a subroutine. This subroutine changes the character set pointer to the user defined graphics, checks the screen, "frigs" the CHR$ value then resets the pointer to the ROM character set (if you're using another character set you'll have to preserve the original value in a variable). X is the PRINT position across the screen and Y is the PRINT position down the screen.

```
8000 POKE 23606,PEEK 23675: POKE
 23607,PEEK 23676-1
8010 LET A$=SCREEN$ (Y,X)
8020 IF A$<>"" THEN LET A$=CHR$
(CODE A$+112)
8030 POKE 23606,0: POKE 23607,60
8040 RETURN
```

With this routine, any blank spaces on screen are not recognised properly unless one of the user defined graphics happens to resemble a space or there is something resembling a space in RAM above the user defined graphics. This is a nuisance as spaces are used a lot — any blank part of the screen is a space after all. There are two ways out of this. Either define a UDG as a space or add to the routine like this:

```
8000 LET A$=SCREEN$ (Y,X): IF A$
=" " THEN RETURN
8010 POKE 23606,PEEK 23675: POKE
 23607,PEEK 23676-1
8020 LET A$=SCREEN$ (Y,X)
8030 IF A$<>"" THEN LET A$=CHR$
(CODE A$+112)
8040 POKE 23606,0: POKE 23607,60
8050 RETURN
```

To make SCREEN$ recognise the block graphics CHR$ 128 to CHR$ 143 is not so easy. There are no dot patterns as such for creating this since they are "calculated" when they need to be printed. You could set up the dot patterns somewhere and

change the character set pointer to point to them or you could define a user defined graphic as the block graphics in turn and use this to check against SCREEN$. Remember that although there are 16 block graphics, only 8 of them need to be checked since 8 are inverse versions of the other 8 and SCREEN$ can cope with inverses as this statement (entered in direct mode or as a program line) will show.

```
CLS:PRINT CHR$ 143:PRINT SCREEN$
  (0,0),CODE SCREEN$ (0,0)
```

CHR$ 143 is GRAPHICS SHIFT 8 or a block of INK dots (an inverse space). SCREEN$ recognises this as an ordinary space CHR$ 32. Of course, an easy way out of this is to copy the entire character set from the ROM into RAM and redefine some of the characters that SCREEN$ can recognise. The characters redefined would normally be the little used ones like the square and curly brackets and other symbols not used much in listings.

NON-DELETABLE PROGRAM LINES

Wouldn't it be nice to be able to insert a line like:

```
10 REM © Fred Bloggs 1982
```

into your program knowing it couldn't be edited out and prevent other people copying that program without your author credit? Deleting the above line is easy: just type 10 followed by ENTER and the line has been deleted in the normal way. What is needed is a method of inserting lines into a listing which are difficult, if not impossible, to delete. Part 1 of the answer is that if you manage to get a line number 0 into a listing it cannot be deleted in any of the normal ways since line number 0 is usually associated with direct commands (e.g. enter the direct command PRINT without a line number: you should get report 0 OK 0:1 meaning everything OK in the first statement of line 0). If you attempt to enter:

```
0 REM © Fred Bloggs 1982
```

you would be rewarded with the cheery message Nonsense in BASIC with report C. So that's out.

What has to be done is to enter a line with a normal line number (e.g. 10) then change this number to a zero. Difficult? Not a bit (no pun intended). We could do this by looking through the program for the line number, followed by a REM a little further on, then POKE away until we got what we wanted. However, this would be very slow and messy. A better way is to use the system variable NXTLIN contained in 23637/8 which contains the address of the start of the next program line (note: line not statement). The Spectrum manual informs us that each BASIC line starts off with a line number stored in two bytes in the order More Significant Byte (MSB) followed by Less Significant Byte (LSB). Therefore, line 1 would be 0,1 and line 258 would be 1,2 (1 * 256 + 2). So, if we POKEd 0 into both bytes we'd get our objective of a virtually undeletable program line. Here's how to do this in a line of BASIC:

```
    1 LET a=PEEK 23637+256*PEEK 2
3638: POKE a,0: POKE a+1,0: STOP

    2 REM © Fred Bloggs
```

RUN the program. Now LIST the program and note the zero line number where line 2 used to be and note also how the line numbers have not been sorted into the correct order; sorting only takes place when they are actually entered into the listing — once they're in they stay in order but any lines entered from then on will go in the correct place. Line 1 is no longer needed — delete it as normal to prevent others using it to undo what you've done. You should now have:

```
0 REM © Fred Bloggs 1982
```

Try deleting it with EDIT; try typing in its line number. Quite secure isn't it? To delete it you will have to go through all that POKEing again. But if you think about it you realise you've got a problem — you can't use the system variable NXTLIN again because line 0 is now the first program line — any other lines entered go through the sorting and will go *after* line 0. NXTLIN will only give the correct address for POKEing if used *before* the line to be POKEd — hard luck. For security I'll leave you to work out how to delete line 0. There are several ways of doing this, all of them rather "roundabout" and not too obvious. There are no prizes for doing this as it is not meant to be done. You could place this all into any part of a program and if you're keen enough, you could place a bright, flashing, coloured copyright statement into each page of a listing so it stands out whichever part is viewed. There is nothing more annoying than a glaringly obvious deterrent like this and the knowledge that you can't get rid of it!

If you aim to use it a lot, you could place the routine on tape and MERGE it into your programs. If starting from scratch you could even use:

SAVE "copyright" LINE 1

which would only leave you with the task of deleting line 1 since the LINE instruction will make the program start itself to create line 0. Alternatively, just save line 0 by itself on tape then use MERGE to add that as the first line of a program. MERGE will quite happily handle the zero line number. When creating this copyright line 0, I suggest you give it PAPER bright white, INK black, FLASHing. This really stands out.

Incidentally, Ø is not the only fun line number that can be POKEd. Try POKEing something like 5Ø and Ø in line 1. Why is the cursor before the line number and facing the wrong way? Actually it's a line number higher than 9999 which the interpreter cannot decode properly; the less than symbol is used instead of a number. For even more funny results try POKEing 64 and Ø in line 1 — where did the program go? Any line number greater than 16383 causes the program not to be listed as the LIST will not go beyond line 16383. The program is still there but just cannot be seen as you'll find out if you POKE things back to normal.

"PRESS ANY KEY TO CONTINUE"

A common requirement is to suspend execution of a program pending an instruction from the operator. An example would be displaying a list of instructions then ask the operator to press any key to continue after finishing reading the instructions. This part of the program may well look like this:

```
.....(instructions)

1000 PRINT "Press any key except
 the shift  keys to continue"
1010 IF INKEY$="" THEN GO TO 101
0
```

This is fine, but if you press CAPS SHIFT or SYMBOL SHIFT the program will ignore you and other people will remark "what a stupid program". There are some ways out of this:

```
.....(instructions)

1000 PRINT "Press any key to con
tinue."
1010 IF INKEY$="" THEN GO TO 101
0

1000 PRINT "Press ENTER to conti
nue"
1010 INPUT A$
```

Incidentally, you may have noticed with the programs using INKEY$, that INKEY$ does not respond to either SHIFT alone, but if both SHIFT keys are pressed simultaneously, the program continues. Pressing both SHIFT keys simultaneously (as when you enter E mode) produces CHR$ 14.

The above examples are fine, but wouldn't it be nice if we could truly press any key to continue? Any key, of course, meaning any of the forty keys on the Spectrum keyboard including both SHIFTs. Here is one way in which this could be done:

```
1000 PRINT "Press any key to con
tinue"
1010 IF INKEY$="" AND IN 65278=2
55 AND IN 32766=255 THEN GO TO 1
010
```

The keyboard is located in what is called I/O space, meaning INPUT/OUTPUT. These are methods of getting information in and out of the computer from and to the outside world. The MIC and EAR sockets, the internal loudspeaker, the keyboard, the printer and microdrives and the RS232 interfaces are all examples of I/O in action. The most significant difference between memory addressing as far as the user is concerned is that PEEK and POKE only work with memory, be it RAM or ROM. The I/O commands IN and OUT are concerned with getting information to or from the computer from or to the outside world. There are 65536 of these I/O ports, just as there can be 65536 memory locations, but they may or may not all be in use, just as all memory space is not used in a 16K Spectrum.

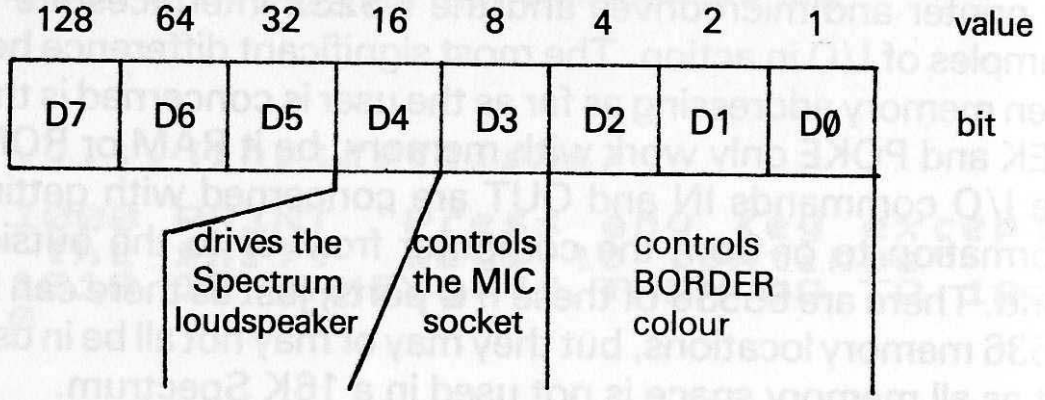
There are two commands in BASIC to handle the I/O ports. These are IN and OUT which can be thought of as working like PEEK and POKE respectively. Functions like INKEY$ also access the I/O ports, but make use of machine code equivalents of IN and OUT in their own little ways. The next question is how do you know which of these ports are used for what? Chapter 23 of the Spectrum manual outlines them briefly, but the ones most likely to be of use are those associated with the keyboard, at least at this stage.

As an example, using OUT, let us play with PORT 254 which amongst other things sets the BORDER colour and drives the loudspeaker. This can be demonstrated by RUNning this short program:

```
10 OUT 254,INT (RND*256)
20 GO TO 10
```

You should hear a clicking noise from the Spectrum's loudspeaker and see the screen's BORDER colour go haywire! The colour changes so rapidly, you may be able to see several BORDER colours at once! Note that whilst this program is running, the lower two lines of the screen do not change colour (they would normally be the same colour as the BORDER). The BORDER reverts to the colour of the lower screen when you

type something. If you understand anything about binary, this diagram of the eight bits of PORT 254 may help to explain how the port manages to do more than one thing at a time. Like a memory location I/O ports are eight-bit bytes.



D0,D1,D2, etc., mean bit 0, bit 1, bit 2, etc. The D usually stands for DATA, but that need not bother us now. Since only bits 0 to 4 are used, we should have replaced line 10 in the previous program with:

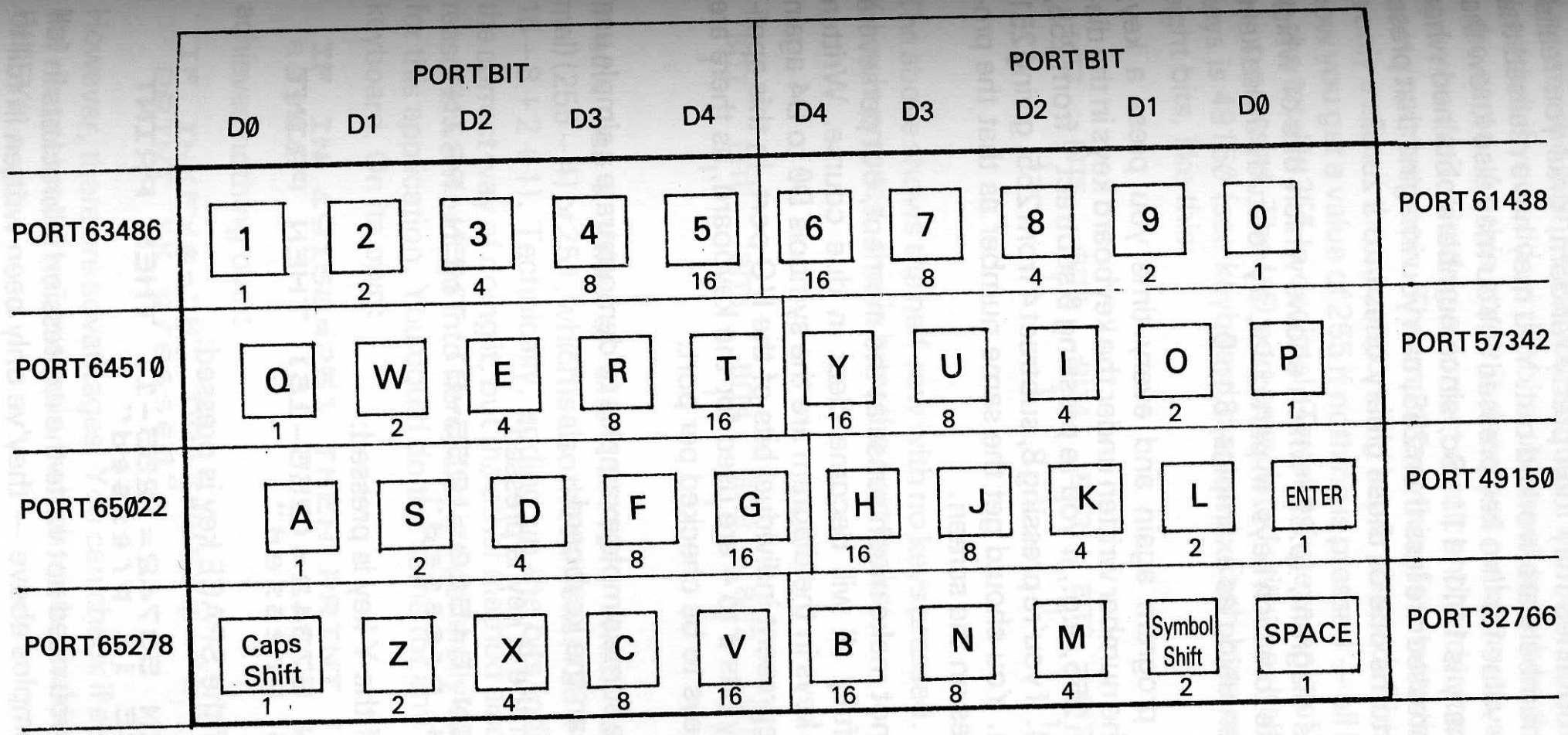
```
10 OUT 254,INT (RND*32)
20 GO TO 10
```

since the bits used could add up to 0 (lowest) and 31 (highest).

More useful to us are the I/O ports that are associated with the keyboard. There are eight ports, each handling a row of five keys on the left or right half of the keyboard. For example, PORT 61438 is associated with the row of five keys, 6 (bit 4), 7 (bit 3), 8 (bit 2), 9 (bit 1) and 0 (bit 0). Try this program which prints out the value of port 61438 over and over again. Try pressing keys 6 to 0 to see what effect it has. Press more than one key at a time.

```
10 PRINT IN 61438
20 PAUSE 100
30 GO TO 10
```

RUN it and see how it prints 255 all the time, unless you press one of the 5 keys in the half-row 6 to 0 on the keyboard. The



values to be subtracted from 255 if key pressed.

Diagram of I/O ports associated with the keyboard and which bit of the ports are associated with each key. Note how bit 0 is always on the outside and bit 4 is nearest the middle of the keyboard.

numbers obtained may look pretty random, until you realise how the numbers are worked out. You may have realised that 255 is the value for "no key pressed". You may also know that 255 in binary is 11111111. So, since numbers obtained when keys are pressed are less than 255, can you imagine that pressing a key turns one of those binary ones into a zero?

Study this diagram, (p.89) which shows which bits of which ports relate to which keys. In particular, try to study those keys we've been using as examples, 6 to Ø.

RUN the program again and every time you press a key, subtract the number written under the keyboard keys in the diagram from 255, e.g., if you're pressing Ø subtract 1 from 255, giving 254. If you're pressing 8, subtract 4 from 255, giving 251 and so on. You should get the same number as that the program writes on the screen.

This may not make much sense at the moment, but persevere and hopefully all will become clear in due course. Written above the keys in the diagram are the symbols DØ to D4 again — these represent individual bits of the I/O port. In this application only bits Ø to 4 are used for the keyboard, as there are only five keys to be checked per port.

Let us have some simple examples to demonstrate a simple use of IN to scan the keyboard.

To check if the R key is pressed:

```
IF IN 64510=(255-8) THEN PRINT
"R is pressed"
```

To check if the Y key is pressed:

```
IF IN 57342=(255-16) THEN PRINT
"Y is pressed"
```

To check if the SPACE key is pressed:

```
IF IN 53742=(255-1) THEN PRINT
"SPACE is pressed"
```

Of course, you need not write the expression in brackets in full like the examples above — they've only been written in full to

illustrate the point that you subtract the bit value from 255. Note that if you add up the bits' values all together, the answer is the same. At this stage it does not make much difference how you do it. Getting a correct result and understanding it is most important now. The important thing is to note that any bit is only a zero if the corresponding key is pressed. This explains how you get a value of 255 if nothing is pressed — all bits are 1, so the total is 255 in decimal. To take the example of the K key being pressed. The I/O port associated with that half-row of 5 keys is 49150 (see keyboard diagram). Each byte or port has eight bits, like this:

BIT7	BIT6	BIT5	BIT4	BIT3	BIT2	BIT1	BIT0
1	1	1	1	1	1	1	1

The above shows the half-row with no keys pressed. When the K key is pressed, this is how the port looks:

BIT7	BIT6	BIT5	BIT4	BIT3	BIT2	BIT1	BIT0
1	1	1	1	1	0	1	1

It now has a value of BIN 11111011 which is (in decimal) (255 − 4) or 251, which is also the same as (128 + 64 + 32 + 16 + 8 + 2 + 1). Technically, adding up the bits individually is the correct way of doing it, but the other method also works for reasons we won't go into here, and it's generally easier to use for this application. You could do the same for any key on the keyboard. On its own:

```
IF IN 49150=251 THEN PRINT "K pr
essed"
```

achieves nothing over:

```
IF INKEY$="K" OR INKEY$="k" THEN
 PRINT "K pressed"
```

However, there are advantages. You can check if either SHIFT key is pressed, for example, which you couldn't do with INKEY$, e.g.:

```
IF IN 65278=254 OR IN 32766=253
THEN PRINT "SHIFT pressed"
```

INKEY$ also differentiates between upper and lower case letters so that IF INKEY$ = "k" THEN... is not the same as IF INKEY$ = "K" THEN... whereas IF IN 49150 = 251 THEN... just checks if the k key is pressed, irrespective of whether CAPS LOCK or CAPS SHIFT is on.

Using IN to scan the keyboard also allows us to check if more than one, or combinations of, keys are being pressed, e.g.:

```
IF IN 49150=(255-2-4) THEN PRINT
 "K and L pressed"
```

One application for this would be in games where the cursor control keys are used to control movement on the screen in the direction of the arrows. Most games only allow you to move left, down, up or right, never diagonally. Using IN we could check to see if both the 5 and 6 keys are pressed to enable movement both left and down, i.e. diagonally towards the bottom left of the screen so that movement control could be more like that of a joystick. Try this program to draw a line going up and right from the bottom left corner towards the top right-hand corner, rather like a graph. The controls are 7 to move up, 8 to move right and press both 7 and 8 to move diagonally up and right. This would not have been so easy if we had used INKEY$ since we would not have been able to check if both the 7 and the 8 key were pressed.

```
10 LET X=0
20 LET Y=0
30 PLOT X,Y
40 LET A=IN 61438
50 LET X=X+(A=251 OR A=243)
60 LET Y=Y+(A=247 OR A=243)
70 GO TO 30
```

Whilst on the subject of using the cursor keys 5, 6, 7 and 8 to control screen movements, wouldn't it be nice if this could be made easier to use? The reasons they are commonly used for this purpose are that they have direction arrows marked near them on the keyboard and they are easy to read with INKEY$ to control variable values (you may be familiar with, say, LET X=X+ (INKEY$="8")-(INKEY$="5")). The snag is that these keys are so close together that it requires some pretty

nimble finger-action for fast, accurate control. The system to be described allows the use of the entire 40-key keyboard to control movement so that you don't have to worry so much about your finger being on the exact key required. The keyboard will be split into 4 (from the point of view of the program, not sawing it apart!) parts, each a block of 10 keys like this controlling movement in the directions shown.

1			UP		0
Q		LEFT T	Y	RIGHT	P
A		LEFT G	H	RIGHT	ENTER
CAPS SHIFT			DOWN		SPACE

So, pressing any of the keys on the top row of the keyboard causes movement upwards, pressing any of the keys on the bottom row of the keyboard causes movement downwards. Pressing keys on the left-hand half of the middle rows of the keyboard makes you move left and pressing keys on the right half of the middle rows of the keyboard makes you move right. Pressing different groups of keys have a combined effect, e.g. if you pressed the 3 key and the W key you would move diagonally up and left. The program to demonstrate the routine is a very simple sketcher program which draws in the direction you're "steering" it. If you're not pressing any keys you stay still, as you'd expect. Do not expect this to be the best sketcher ever — it crashes if you go off the edge of the screen! Refer

back to the diagram showing the I/O ports associated with the keyboard when examining lines 30 and 40 which do all the keyboard scanning.

```
   10 LET X=120
   20 LET Y=90
   30 LET X=X+(IN 49150<>255 OR I
N 57342<>255)-(IN 64510<>255 OR
IN 65022<>255)
   40 LET Y=Y-(IN 65278<>255 OR I
N 32766<>255)+(IN 63486<>255 OR
IN 61438<>255)
   50 PLOT X,Y
   60 GO TO 30
```

PRINTING STRING ARRAYS

Try this program:

```
10 DIM A$(12,9)
20 FOR A=1 TO 12
30 READ A$(A)
40 PRINT A$(A);",";
50 NEXT A
60 PRINT CHR$ 8;"."
70 DATA "January","February","
March","April","May","June","Jul
y","August","September","October
","November","December"
```

What it should do is assign the names of the months of the year to each string of the array (12 months, maximum length of 9 letters — DIM A$(12,9)) and print them out end to end in a row, separated by commas, ending with a full stop. This is how it should appear:

```
January  ,February ,March    ,Ap
ril    ,May      ,June     ,July
     ,August   ,September,Octobe
r   ,November ,December .
```

Some of the words are separated by several spaces when printed because each string in the array A$ is always the same length (in this case nine characters). Each time you assign to one of these strings, you assign to all characters of the string, even if the computer has to add extra spaces. For instance, if you're making A$(5) MAY which consists of only three letters, then six spaces must be added to make up the nine characters. The result is a lot of ugly spaces printed along with the required letters which looks messy on the display. This problem can be overcome in two main ways.

(1) Print the string, scanning backwards to find the end of the word, like this:

```
10 DIM A$(12,9)
20 FOR A=1 TO 12
30 READ A$(A)
40 REM print,omit end spaces
50 FOR B=LEN A$(A) TO 1 STEP -
1
60 IF A$(A,B)<>" " THEN PRINT
A$(A, TO B);",";: GO TO 80
70 NEXT B
```

```
  80 NEXT A
  90 PRINT CHR$ 8;"."
 100 DATA "January","February","
March","April","May","June","Jul
y","August","September","October
","November","December"
```

This will be the result of RUNning the program:

```
January,February,March,April,May
,June,July,August,September,Octo
ber,November,December.
```

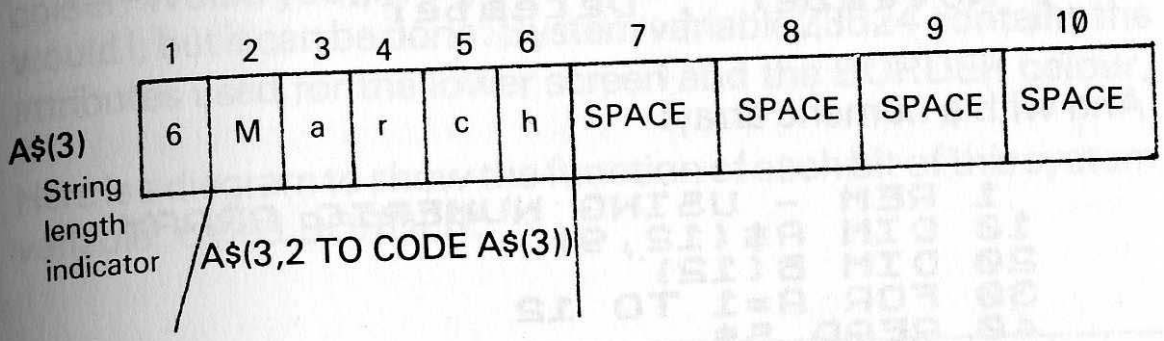
All the program does is read the names of the twelve months into the string array A$(12,9) and the printing is done in the FOR-NEXT loop B which works backwards from the last character in each string towards the first until it finds a character that is not a space. It then prints to this point with A$(A, TO B). Note the comma. This expression is an abbreviation of A$(1 TO B).

(2) Use a String Length Indicator (SLI) to record the length of the words in the strings so that we do not need to waste time scanning before printing. This method also allows words ending in spaces (if needed) to be printed correctly. The SLI will normally be stored as the first character in each string. Here is a listing to demonstrate:

```
  10 DIM A$(12,10)
  20 FOR A=1 TO 12
  30 READ S$
  40 LET A$(A)=CHR$ (LEN S$+1)+S
$
  50 PRINT A$(A,2 TO CODE A$(A))
;",";
  60 NEXT A
  70 PRINT CHR$ 8;"."
  80 DATA "January","February","
March","April","May","June","Jul
y","August","September","October
","November","December"
```

Note how in line 10 an extra element has been added to each string of the array A$ — to store the SLI. The number representing the length of the string may only be a whole number in the range 0 to 255. This should be enough for most applications. Two bytes could be used if larger numbers are required up to 65535. The loop A reads the names of the months from the data list into an ordinary common-or-garden string variable S$. Since string variables are only as long as they need be, we

can use LEN S$ to find the length of the word and store this in the first character of the strings of the array A$ as CHR$(LEN S$ + 1). This is the extra element we talked about earlier. The rest of the string is made into a copy of S$. This is how it would look for the word March:



So, the first element of the string A$(3) contains a character 6. The length of the word March is 5 letters. It starts at the second element and ends at element 6. The reason for adding 1 to the length of March was because the SLI is the first character — it would be unwise to place it at the end in case the string assigned was so long it overwrote the SLI. The one problem is that the SLI may be greater than the length of the strings of A$ thus generating a subscript error. This could be avoided by adding the line:

```
   45 IF CODE A$(A)>=LEN A$(A) TH
EN LET A$(A,1)=CHR$ (LEN A$(A)-1
)
```

The strings are printed as PRINT A$(A,2 TO CODE A$(A)), meaning from after the SLI to the last character which isn't a space. The one big advantage of method (2) over method (1) is that you can correctly print words ending with spaces and if you want to copy strings into other strings or just generally muck them about, storing the SLI enables it to be done quickly.

If you would like the speed and convenience of SLIs but without the rather messy strings, then it is a simple matter to store the SLIs in a separate string array or numeric array. String array first:

```
  1 REM - USING STRING ARRAY -
 10 DIM A$(12,9)
 20 DIM B$(12)
 30 FOR A=1 TO 12
 40 READ S$
```

```
   50 LET A$(A)=S$
   60 LET B$(A)=CHR$ (LEN S$)
   70 PRINT A$(A, TO CODE B$(A));
",";
   80 NEXT A
   90 PRINT CHR$ 8;"."
  100 DATA "January","February","
March","April","May","June","Jul
y","August","September","October
","November","December"
```

And with a numeric array:

```
    1 REM - USING NUMERIC ARRAY -
   10 DIM A$(12,9)
   20 DIM B(12)
   30 FOR A=1 TO 12
   40 READ S$
   50 LET A$(A)=S$
   60 LET B(A)=LEN S$
   70 PRINT A$(A, TO B(A));",";
   80 NEXT A
   90 PRINT CHR$ 8;"."
  100 DATA "January","February","
March","April","May","June","Jul
y","August","September","October
","November","December"
```

You should have found that using a numeric array is faster but uses more memory. Again, this could be turned into a sub-routine for use within programs.

LOWER SCREEN ATTRIBUTES

Normally you cannot change the attributes of the lower screen, except for the PAPER colour, which follows the BORDER colour. Would you like a green report code and cursor? Neither would I, but it can be done. System variable 23624 contains the attributes used for the lower screen and the BORDER colour.

Here is a diagram to show the function of each bit of this system variable, called BORDER:

bit	7	6	5	4	3	2	1	0
	FLASH lower screen	lower screen BRIGHT	BORDER colour and lower screen PAPER			lower screen INK		

By POKEing various values into this system variable you could achieve say a flashing black and white lower screen or a white lower screen that stands out brighter than the rest of the white screen for INPUTs, etc. Try these two with a white screen (PAPER 7:CLS):

POKE 23624,BIN 10111000 (184)
POKE 23624,BIN 01111000 (120)

You cannot normally get this effect with INPUT statements as colour controls, etc., only effect the prompt string. Changing BORDER colour affects this; for instance, lower screen INK (being "automatically" 9) would revert to either white or black to ensure maximum contrasts so that anything typed in the lower screen can be read easily.

PREVENTING AUTO RUNNING

If a program has been saved on tape using the SAVE "prog" LINE X facility, that program will start automatically when loaded back into the computer, starting from line X. To prevent this occurring if you so wished for any reason MERGE "" can be used. Ensure there was no program in the computer before using MERGE "" in this way, as you may get a combination of both programs.

SPEEDING UP YOUR PROGRAMS

Here are some tips to help you ensure your Spectrum BASIC programs run as fast as possible. Using these you may be able to turn a jerky graphics program into a smoother, more acceptable version. By far and away the worst programs are the messy, poorly laid out ones. A long tangle of convoluted GOTOs all over the place will often slow your program down as much as will a few PAUSEs! The first thing to do is tidy up your programs so that you can identify whole blocks as separate routines. Not only does this make it easier to read the program, but you can test or alter a particular routine without affecting or depending upon the others. But most of all the program will not waste time jumping backwards and forwards unless it really has to. Giving the program a good *structure* as it's called often works wonders. Examine any non-conditional GOTOs critically and try to make the program built up around a series of subroutines called from a main block or loop like this:

```
   10 REM main program

 ............

4000 STOP
5000 REM initialise
5900 RETURN
6000 REM set up graphics
6900 RETURN
          etc. etc.
```

The point to note here is that when searching for GOTO or GOSUB destinations, it is necessary for the computer to start at the beginning of the program and work up through the line numbers until the one sought is found. This means that if you called a subroutine at the end of a long program several times during the course of the program, it would take longer than if that subroutine was at the beginning of the program. We can use these simple programs to illustrate this:

PROGRAM 1 — 9 seconds

```
10 FOR A=1 TO 1000
20 GO SUB 9000
30 NEXT A
```

```
   40 STOP
   50 REM
   60 REM
   70 REM
   80 REM
   90 REM
  100 REM
  110 REM
  120 REM
  130 REM
  140 REM
  150 REM
  160 REM
  170 REM
  180 REM
  190 REM
  200 REM
 9000 RETURN
```

PROGRAM 2 — 7·5 seconds

```
   10 FOR A=1 TO 1000
   20 GO SUB 9000
   30 NEXT A
   40 STOP
 9000 RETURN
```

Running program 1 should take about 9 seconds compared with about 7·5 seconds for program 2. The reason program 1 takes longer is that to find the subroutine at line 9000 it has to skip over all the lines one by one. Note that the line number itself has no effect on speed, only its position within the program. From this we can decide that programs should be laid out so that the most commonly used GOTO and GOSUB destinations should be positioned like this:

```
   1 REM suggested program layout
  10 GOTO 1000: REM skip over subroutines
 100 REM frequently used GOSUBs, etc.
1000 REM main program
8000 REM infrequently used GOSUBs, etc.
```

This sort of layout would only be used where the time saved is important. As with most of these time saving techniques they only become obvious during large loops. Then again, this is where you are most likely to need to save time. As we're talking about loops, there are two main types of loops that perform a similar function — the FOR/NEXT loops and the IF...THEN GOTO... loops. Let us compare the speed of both.

PROGRAM 3 — 9 seconds

```
10 LET A=1
20 LET A=A+1
30 IF A<=1000 THEN GO TO 20
```

PROGRAM 4 — 4·5 seconds

```
10 FOR A=1 TO 1000
20 NEXT A
```

So the FOR/NEXT loop is approximately twice as fast as its IF...THEN GOTO counterpart. As an interesting comparison try comparing program 4 with program 5 which uses multiple statements.

PROGRAM 5 — 4·5 seconds

```
10 FOR A=1 TO 1000: NEXT A
```

Note that unlike most versions of BASIC, Spectrum BASIC does not offer much time saving when using multiple statements compared with the same program written with all statements on individual line numbers. Program 5 takes about the same time to RUN as program 4. Within PRINT statements though, stringing several statements on one line can run faster than on separate lines.

PROGRAM 6 — 45 seconds

```
10 FOR A=1 TO 1000
20 PRINT AT 0,0;"■■INKING"
30 PRINT AT 0,0;"TH■■KING"
40 PRINT AT 0,0;"THIN■■NG"
50 PRINT AT 0,0;"THINKI■■"
60 NEXT A
```

PROGRAM 7 — 41 seconds

```
10 FOR A=1 TO 1000
20 PRINT AT 0,0;"■■INKING";AT
0,0;"TH■■KING";AT 0,0;"THIN■■NG"
;AT 0,0;"THINKI■■"
30 NEXT A
```

When you want to move the PRINT position to a new line, using the apostrophe rather than TAB Ø; is faster.

PROGRAM 8 — 23 seconds

```
10 FOR A=1 TO 300
20 POKE 23692,255
30 PRINT "+";TAB 0;
40 NEXT A
```

PROGRAM 9 — 18 seconds

```
10 FOR A=1 TO 300
20 POKE 23692,255
30 PRINT "+";
40 NEXT A
```

Using the control character CHR$ 13 (the ENTER character) to force a carriage return in place of the apostrophe does not produce any improvement in run times. The inclusion of control characters in strings can slow things down somewhat. Program 10 fills the screen with + signs. Program 11 fills the screen with inverse + signs which should be entered as "INV. VIDEO + TRUE VIDEO".

PROGRAM 10 — 4·7 seconds

```
10 FOR A=1 TO 704
20 PRINT "+";
30 NEXT A
```

PROGRAM 11 — 5·5 seconds

```
10 FOR A=1 TO 704
20 PRINT "■";
30 NEXT A
```

However, compare program 11 with program 12, which uses the INVERSE facility. This it turns out is slower than including the inv. video control character in the string. Experiment with others such as FLASH or BRIGHT in this way.

PROGRAM 12 — 6·2 seconds

```
10 FOR A=1 TO 704
20 PRINT INVERSE 1;"+";
30 NEXT A
```

Some computers have special variables called integer variables which can only store whole numbers. They can be handled much more quickly than normal floating point variables. There are no such integer variables on the Spectrum but it does have

a special representation for small integers which is used automatically for numbers that lie from −65535 to +65535 and are whole numbers.

PROGRAM 13 — 11·3 seconds

```
 5 POKE 23692,255
10 FOR A=1 TO 600
20 PRINT A;
30 NEXT A
```

PROGRAM 14 — 15·3 seconds

```
 5 POKE 23692,255
10 FOR A=1.3 TO 600.3
20 PRINT A;
30 NEXT A
```

Programs 13 to 18 attempt to show how using integers can make programs run faster. You have no control over this "automatic" integer action so the best you can do is make sure you don't generate non-integer numbers which slow things down without you realising it's happening.

PROGRAM 15 — 12·1 seconds

```
 5 POKE 23692,255
10 FOR A=1 TO 600
20 PRINT 1.3;
30 NEXT A
```

PROGRAM 16 — 9·1 seconds

```
 5 POKE 23692,255
10 FOR A=1 TO 600
20 PRINT 1;
30 NEXT A
```

PROGRAM 17 — 12·3 seconds

```
 5 POKE 23692,255
 7 LET B=1.3
10 FOR A=1 TO 600
20 PRINT B;
30 NEXT A
```

PROGRAM 18 — 9·2 seconds

```
 5 POKE 23692,255
 7 LET B=1
10 FOR A=1 TO 600
20 PRINT B;
30 NEXT A
```

You should attempt to use integers where possible as they can be printed or converted to another format more quickly than ordinary floating point numbers. Remember that constants and variables can be held in this format. Also, the amount of scrolling of the screen that has to be done in the programs above have a small effect on run times since the bottom of the screen is reached more quickly with numbers with more digits, and floating point numbers will have more digits than their integer counterparts.

To stay with screen printing let us look at the ways of printing numbers to the screen. Programs 19, 20 and 21 compare printing string constants, integer numeric constants and non-integer numeric constants.

PROGRAM 19 — 9·8 seconds

```
 5 POKE 23692,255
10 FOR A=1 TO 1000
20 PRINT "69";
30 NEXT A
```

PROGRAM 20 — 17·2 seconds

```
 5 POKE 23692,255
10 FOR A=1 TO 1000
20 PRINT 69;
30 NEXT A
```

PROGRAM 21 — 24·8 seconds

```
 5 POKE 23692,255
10 FOR A=1 TO 1000
20 PRINT 69.25;
30 NEXT A
```

Let us now do the same for variables. Programs 22, 23 and 24 print string variables, integer numeric variables and non-integer numeric variables respectively.

PROGRAM 22 — 10·3 seconds

```
 5 POKE 23692,255
 7 LET B$="69"
10 FOR A=1 TO 1000
20 PRINT B$;
30 NEXT A
```

PROGRAM 23 — 17·9 seconds

```
  5 POKE 23692,255
  7 LET B=69
 10 FOR A=1 TO 1000
 20 PRINT B;
 30 NEXT A
```

PROGRAM 24 — 25·2 seconds

```
  5 POKE 23692,255
  7 LET B=69.25
 10 FOR A=1 TO 1000
 20 PRINT B;
 30 NEXT A
```

Again, the message is — use strings for printing wherever possible. If it has to be a number, make sure this is an integer as this will be printed more quickly. Let us look at VAL and CODE as means of printing.

PROGRAM 25 — 26·1 seconds

```
  5 POKE 23692,255
 10 FOR A=1 TO 1000
 20 PRINT VAL "69";
 30 NEXT A
```

PROGRAM 26 — 18·3 seconds

```
  5 POKE 23692,255
 10 FOR A=1 TO 1000
 20 PRINT CODE "E";
 30 NEXT A
```

So where you want to store information as characters in a string, it would be better to decode using CODE than using VAL.

The use of multiple character variable names is slightly slower than single character variable names.

PROGRAM 27 — 6·9 seconds

```
 10 FOR A=1 TO 1000
 20 LET M=10
 30 NEXT A
```

PROGRAM 28 — 7·8 seconds

```
10 FOR A=1 TO 1000
20 LET MAGAZINE=10
30 NEXT A
```

Define the most often used variables first; the time to find a variable in the variables area depends on how far the computer has to look through this area.

PROGRAM 29 — 8·4 seconds

```
10 LET B=10
20 LET C=20
30 LET D=4
40 LET E=6
50 LET F=7
60 FOR A=1 TO 1000
70 LET G=8
80 NEXT A
```

PROGRAM 30 — 8·8 seconds

```
10 LET B=10
20 LET C=20
30 LET D=4
40 LET E=6
50 LET F=7
60 FOR A=1 TO 1000
70 LET G=F
80 NEXT A
```

Let us look at REM statements. Although these are ignored during the RUNning of a program, the BASIC interpreter still has to step over them and this takes a finite amount of time. The odd REM statement here or there won't make much difference, but can have a noticeable effect within a large loop. Compare program 4 with program 31:

PROGRAM 31 — 5·2 seconds

```
10 FOR A=1 TO 1000
20 REM This is a REM statement
30 NEXT A
```

Program 4, an empty FOR/NEXT loop takes 4·5 seconds to RUN compared with 5·2 seconds with an inbuilt REM statement. This is true of everything within loops — if it can be put outside the loop, the computer won't waste time doing it over and over again.

Let us now look at the various ways of writing expressions which are used several times during a program. We'll look at the example of LET R = INT (RND * 9) + 1. First by writing it out in full every time it's required.

PROGRAM 32 — 13·3 seconds

```
10 FOR A=1 TO 500
20 LET R=INT (RND*9)+1
30 NEXT A
```

Or by defining a function.

PROGRAM 33 — 13·9 seconds

```
 5 DEF FN R()=INT (RND*9)+1
10 FOR A=1 TO 500
20 LET R=FN R()
30 NEXT A
```

Or by putting the expression in a subroutine.

PROGRAM 34 — 14·9 seconds

```
10 FOR A=1 TO 500
20 GO SUB 50
30 NEXT A
40 STOP
50 LET R=INT (RND*9)+1
60 RETURN
```

Or applying VAL to a string containing the expression.

PROGRAM 35 — 19·6 seconds

```
 5 LET A$="INT (RND*9)+1"
10 FOR A=1 TO 500
20 LET A=VAL A$
30 NEXT A
```

So you can now see that it is fastest to write the expression out in full every time it is needed.

Now for ATTR and SCREEN$. If you have a choice of using either SCREEN$ or ATTR to find out what is at any screen location, it is faster to use ATTR than SCREEN$, as programs 36 and 37 show.

PROGRAM 36 — 9·5 seconds

```
10 FOR A=1 TO 1000
20 LET B=ATTR (1,1)
30 NEXT A
```

PROGRAM 37 — 12·5 seconds

```
10 FOR A=1 TO 1000
20 LET B$=SCREEN$ (1,1)
30 NEXT A
```

MAKING USE OF THE SYSTEM VARIABLES

System variables are bytes in memory from 23552 to 23732 that help the computer remember certain things it needs to know about itself like how its memory is laid out. The information is held in these system variables in these addresses so that the computer can get hold of it and update it as and when required.

We can make use of the information stored in these memory locations, in some ways, in our programs either by reading information already there or changing it to make the computer do something it might not otherwise do, or sometimes do it more easily.

Not all of them are that much use to us. And certainly not all of them ought to be changed. Some will cause the computer to crash, or the computer may simply ignore you. Some can be happily changed under certain circumstances only and most within strict limitations. I hope to give you some guidelines as to what can and can't be done, but hopefully you will learn your own little PEEKs and POKEs in time as well.

23552 to 23559 KSTATE reading the keyboard

When the processor is interrupted (50 times every second in the UK normally) one of the things done is to read the keyboard and store the results here. The bytes have different uses. Not all can be practically used by the programmer. You can use this program to examine what's going on in the eight bytes of KSTATE. Run it and press various keys to see what effect individual keys have, such as the SHIFT keys, and what effect going from one key to the other has.

```
   10 FOR A=23552 TO 23559
   20 POKE 23692,0: REM KEEP SCRO
LLING
   30 LET B=PEEK A
   40 PRINT A;TAB 10;B;TAB 20;CHR
$ B AND B>31
   50 NEXT A
   60 GO TO 10
```

The first four bytes of KSTATE deal with something called "two key rollover" which allows you to press a second key

before you actually let go of the first. The descriptions given to the main four bytes, 23556 to 23559, will apply to the first four also as long as you bear in mind that these only come into operation for two key rollover. PEEK 23556 can return the CODE of the upper case version of the key pressed, so if you pressed SYMBOL SHIFT A you would get the CODE of "A" not the CODE of "a" nor the CODE of "STOP". This may be useful where it is essential that upper case be entered, etc. The effect of pressing a key is temporary and lasts only as long as the key is being pressed. The value in 23556 would be 255 if no key was being pressed at the time the interrupt had occurred. For the ENTER key a value of 13 is returned. For the SPACE key a value of 32 is returned. Pressing both SHIFT keys simultaneously produces 14. This program will demonstrate this:

```
10 LET A=PEEK 23556
20 POKE 23692,0
30 PRINT A,CHR$ A AND A>31
40 GO TO 10
```

23557 is used for timing to prevent intermittent key contact, etc., causing problems — known as keyboard debouncing.

23558 is the auto repeat timer which times the pause before the keys start repeating, then the pause between repeats once the key has actually started repeating. The delays used are those in the system variables that hold these delays (23561/2).

23559 contains the CODE of the last character pressed on the keyboard. This depends on whether the SHIFT keys were pressed or not. The numbers produced are as those that would be returned by PRINT CODE INKEY$ except that these are the last key pressed not necessarily the key currently being pressed. Try this program to display what can happen — RUN it and try pressing various keys making use of the SHIFT keys.

```
10 LET A=PEEK 23559
20 POKE 23692,0
30 PRINT A,CHR$ A AND A>31
40 GO TO 10
```

See also under 23611 FLAGS.

23560 LASTK Newly pressed key

Every time the keyboard is scanned and a key is found to have been pressed and proved valid the value of this system variable

is updated. Its content is the CODE of the last key pressed. It does not really do much you could not do with INKEY$ except that it could be used to type ahead one character. If you try this program, you will find that if you press a key when invited to do so, the key is indicated on the screen in a short while even though the program may not have got as far as line 50 when you pressed a key. The CODE of the last key pressed is stored here and stays here until another key is pressed. It is possible to test for a newly pressed key by examining bit 5 of the system variable FLAGS 23611. This would be 1 for a key just pressed.

```
10 PRINT "Press a key now"
20 FOR A=1 TO 900
30 NEXT A
40 CLS
50 LET A=PEEK 23560
60 PRINT A: IF A>31 THEN PRINT
CHR$ A
```

This could be used for testing for a y/n (yes or no) type situation — if you knew one was coming up you could indicate your response before the program got there and the program would respond when it got round to it. Also, if two keys were pressed simultaneously the program would respond if one were released without having to wait for the keyboard to be released completely.

Control characters can be generated by using CAPS SHIFT in conjunction with the number keys. ENTER returns 13. Pressing both SHIFT keys together returns 14. To see this, try this program:

```
10 LET A=PEEK 23560
20 PRINT A,CHR$ A AND A>31
30 GO TO 10
```

23561 REPDEL Repeat delay

This system variable contains the length of time that a key must be held down before it starts to auto-repeat. The time delay in the UK is in one fiftieths of a second and starts off at 35/50 of a second. You can happily POKE this if, for instance, you want the key to start repeating immediately. The cursors become rather difficult to control if you, say, POKE 23561,1. POKE 23561,0 effectively turns off the auto-repeat, actually giving a delay of about 5 seconds like POKE 23561,255.

23562 REPPER delay between repeats

This system variable controls the length of time between repeats once the auto-repeat has actually begun. The time is in fiftieths of a second in the UK. If you effectively want to turn off the auto-repeat for any reason, POKE 23562,Ø or POKE 23562,255 gives about 5 seconds between repeats. If you wish to edit long program lines (e.g. a long PRINT statement) then POKE 23562,1 will speed up moving the cursor to the right place. But beware of changing 23561 too much at the same time or you may speed up the cursor so much it becomes difficult to control. Its normal value is 5/5Ø of a second or one tenth of a second.

23563/4 DEFADD

The address of the argument of a user defined function in a program, i.e. if you had DEF FN A(B) in a program line, the value in 23563/4 would be the address of the letter B in the brackets in that line while only the function is being used. The best way to PEEK into 23563/4 to show this is to put the PEEK as a part of the FN to be evaluated as there is always Ø there unless the function is being evaluated. So the line:

```
  10 PRINT PEEK 23563+256*PEEK 2
3564
```

would always return Ø. On the other hand:

```
  10 DEF FN A(B)=PEEK 23563+256*
PEEK 23564
  20 PRINT FN A(999)
```

would return the address of the B in line 1Ø. The 999 is not significant, just something to actually give a value to B to prevent an error. In the case of a function with no argument:

```
  10 DEF FN A()=PEEK 23563+256*P
EEK 23564
  20 PRINT FN A()
```

this would print the address of the close bracket) symbol.

23568 to 23605 STRMS

The first fourteen bytes on a basic Spectrum contain addresses relating to channels and streams. Streams −3 to +3 are stored in two bytes each.

23606/7 CHARS

This system variable has as normal values:

23606 contains 0

23607 contains 60

This system variable points to the start of the character set that the computer uses for printing on the screen and on the printer. SCREEN$ also uses this system variable. The normal address pointed to is 15360 which is 256 less than the address of the start of the ROM character set. 256 less because the character generator is accessed by something similar to PEEK 23606 + 256 * PEEK 23607 + CODE "A" * 8 and since the first character is SPACE, CODE of SPACE is 32, and 8 * 32 is 256. The character generator is 768 bytes long, so if you wish to set up a new character set you must set aside this number of bytes in case it is overwritten by BASIC — you wouldn't get a crash, you'd just end up with gibberish. Mention was made of SCREEN$ using this system variable — in fact, you may be aware of the problem that SCREEN$ does not recognise user-defined graphics normally, unless they happen to be similar to an existing Spectrum character. In fact, SCREEN$ works by picking up the address of the start of the character generator and looking through the table until it finds a matching character. Now since the Spectrum screen is bit-mapped rather than memory mapped like some computers, once anything is printed on the screen it stays the same even if you change the character in memory. So, we could temporarily change the pointer to the character set to point to the user defined graphics and look up there. One snag is that although there is a system variable that tells us where the user defined graphics start, this address is not 256 less — so we must subtract 256. This conveniently means we subtract one from the high byte. This program should demonstrate:

```
10 FOR X=144 TO 164
20 PRINT AT 0,0;CHR$ X
30 POKE 23606,PEEK 23675
40 POKE 23607,PEEK 23676-1
50 PRINT AT 20,0;SCREEN$ (0,0)
60 PAUSE 40
70 POKE 23606,0
80 POKE 23607,60
90 NEXT X
```

What we did was make the computer think the user-defined graphics are the normal character set. SCREEN$ will still produce characters with CODEs of 32–127 though, although this is easily overcome with a bit of fiddling. Since SCREEN$ starts off with CHR$ 32 and the UDGs start off at 144, we would need to add 112 to return characters in the range of the user defined graphics. Here is one way to do this. X is the x-co-ordinate across the screen and Y the y-co-ordinate down the screen of the location SCREEN$ is to examine. A check is first of all made that SCREEN$ does not find one of the normal characters there, then returns if one is found. The character at Y,X is returned in A$. Line 8025 is needed only if you are using a character set other than the ROM one. If you are using the ROM character set, then delete line 8025 and replace lines 8070 and 8080 with the alternative versions that follow.

```
8000 REM SCREEN$ FOR UDG'S
8010 LET A$=SCREEN$ (Y,X)
8020 IF A$<>"" THEN RETURN
8025 LET A=PEEK 23606: LET B=PEE
K 23607
8030 POKE 23606,PEEK 23675
8040 POKE 23607,PEEK 23676-1
8050 LET A$=SCREEN$ (Y,X)
8060 IF A$<>"" THEN LET A$=CHR$
(CODE A$+112)
8070 POKE 23606,A
8080 POKE 23607,B
8090 RETURN

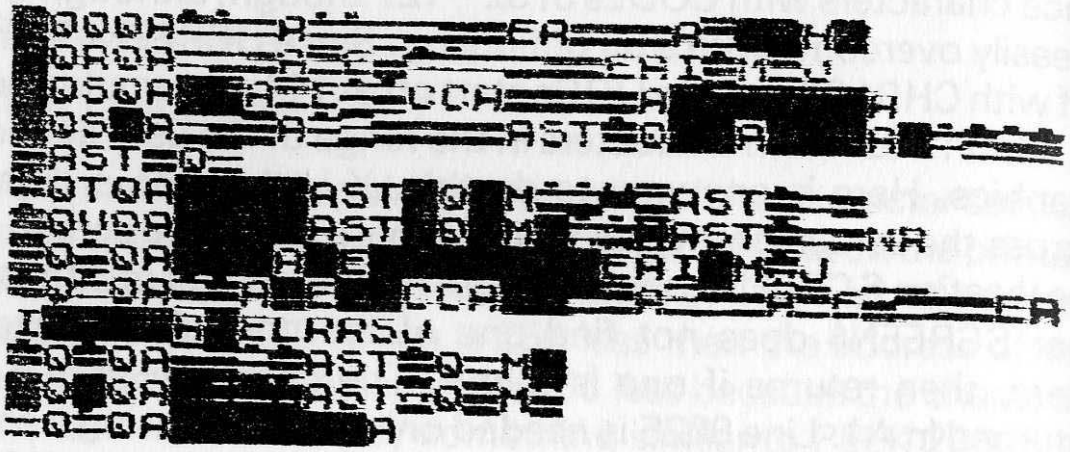
8070 POKE 23606,0
8080 POKE 23607,60
```

The story does not finish there. There are only 21 user defined graphics — if SCREEN$ does not find a match, it will continue looking up past the user defined graphics until it has finished looking for the 32 to 127 range it thinks it's looking for. This could be embarrassing if there just happened to be some data stored above the UDGs for any reason which resembled any character. To help prevent this happening although the UDGs are normally at the top of RAM anyway, this could be added:

```
8065 IF A$>CHR$ 164 THEN LET A$=
""
```

Incidentally, you should ensure that 23606/7 always points to the right character set when PRINTing, LISTing, etc., is done — this is the sort of thing that careless use of this system varia-

ble can do. It's actually one of the above subroutines without the CHARS pointer reset. Not particularly readable is it?



23608 RASP

Controls the duration of the buzz that sounds to warn you that you are running out of memory. At switch on, this has a value of 64. This can be altered, but there seems little point. POKE 23608,0 gives a very short click rather than a buzz especially if you hate that buzz that satirically mocks you when you run out of memory. Alternatively, POKE 23608,255 gives a very long buzz which immobilises the keyboard preventing you typing in any further than when the buzz sounded.

23609 PIP

Controls the length of the click emanated when a key is pressed in command mode or during an INPUT. Starts off at 0 but may be changed. Any value between 30 and about 130 gives a pleasant, more audible bleep rather than the quieter click normally given. Values higher than 130 tend to noticeably slow down the keyboard response, since computing stops as the bleep is sounded. Usually, the one to use is POKE 23609,100

23610 ERR NR

Controls error report number and normally has a value of 255 unless an error arises, when it contains one less than the error report codes printed, e.g. for error 4, out of memory, would contain 3. The message printed out is contained in ROM starting at address 5010 decimal. The end of the message is signified by the last character in the message having bit 7 set to 1. After the error messages comes the © 1982 Sinclair Research

Ltd., message that you see after switch on or NEW. You can POKE 23610 to generate an error to stop the program, but since the message printed is fixed and in ROM you may end up with garbage. If you wanted to simulate an out of memory error you would end a program with POKE 23610,3. This would not work as any program line, you'd have to ensure that it was the last line as once a condition arises to end a program — only then is 23610 looked at as data to determine what is printed.

23611 FLAGS

This system variable contains various flags controlling the BASIC system and generally should not be POKEd. However, some of the flags can be usefully PEEKed.

BIT 0: Being 1 indicates no space to be printed before the next keyword.

BIT 1: This bit being set to 1 indicates print output to be sent to the printer. 0 means send to the TV screen.

BIT 5: Any newly pressed key is indicated by its CODE being stored in 23560 (the LASTK system variable) and bit 5 of 23611 (FLAGS) is set to indicate that a new key has been pressed.

BIT 7 : Syntax flag.

These will be of more use when using ROM routines in a machine code program than to the BASIC programmer.

23613/4 ERR-SP

Keeps track of the address on the machine stack where the appropriate return data lies. Try calling a few GOSUBs with no matching RETURNs and watch this point down the memory. Now you can see what happens and why this occurs when you run out of memory in a situation like this. Also, try PEEKing the contents of the three addresses, the base of which is pointed to by 23613/4, to see what return data actually consists of.

```
  10 LET A=PEEK 23613+256*PEEK 2
3614
  20 PRINT PEEK A;TAB 10;PEEK (A
+1);TAB 20;PEEK (A+2)
```

23617 MODE

Specifies cursor. 0,1,2, or 4 for L/C mode, E mode, G mode or K mode respectively. POKEing this system variable will affect the appearance of the cursor — it may appear as a flashing letter, number, symbol or even a keyword. This is most apparent during an INPUT statement. The value is reset when the need arises, e.g. a mode change made normally from the keyboard. So if you get into difficulties, press both SHIFT keys for E mode and then the same again to get back to normal L/C mode. Try this program which POKEs all possible values into 23617. Most are variants on the four cursors, i.e. you will find yourself in a particular mode after the POKE such as everything coming out as graphics as in G mode. 252 will give an L/C mode flashing < to point to where you're typing . . .

```
10 FOR A=0 TO 255
20 PRINT A
30 POKE 23617,A
40 INPUT A$
50 NEXT A
```

23618/9 NEWPPC and 23620 NSPPC

23618/9 is a two byte system variable containing the line number of the line to be jumped to. 23618 contains the lower byte of the line number and 23619 the higher, so the line number contained is read as PEEK 23618 + 256 * PEEK 23619. To POKE a line number in, say line X:

POKE 23618, X − 256 * INT (X/256)

POKE 23619, INT (X/256)

```
POKE 23618,X-256*INT (X/256)
POKE 23619,INT (X/256)
```

We now come to system variable 23620. With 23618/9 and 23620 we could actually simulate a GOTO to a statement within a program line should that ever be necessary. GOTOs cannot access individual statements within long program lines.

To jump to statement 4 in line X, first go through the motions described above then POKE 23620,4 and the jump is executed.

23624 BORDCR

The bits of this system variable control the attributes of the lower screen and the BORDER colour as follows:

bit	7	6	5	4	3	2	1	0
	FLASH lower screen	lower screen BRIGHT	BORDER colour and lower screen PAPER			lower screen INK		

By POKEing various values into this system variable you could achieve a flashing bright multicoloured lower screen, or make both PAPER and INK the same colour to prevent other people getting at your programs — any alteration would have to be made blind. Or you could make INPUTs extra bright to stand out.

23629/30 DEST

The address of the variable when it is assigned to. If the variable had been set up before, it would point to the start of where it was stored in the variables area. If it was being defined for the first time, it would point to the address of the start of the name of the variable in the program, e.g. in 10 LET A = 5 it would point to the address of the letter A. It can also be used to find the memory address of a numeric variable, if you use something like LET A = A like this:

```
10 LET A=5
20 LET A=A
30 PRINT PEEK 23629+256*PEEK 2
3630
```

23631/2 CHANS

Stores the address of where the channel information area starts.

23633/4 CHURCHL

Address of INPUT/OUTPUT information used at that moment. Normally points during an INPUT/OUTPUT operation to a 5 byte block of data in the channel information area. Use this to examine the contents.

```
  1 FOR X=0 TO 3: PRINT #X;PEEK
 23633+256*PEEK 23634: NEXT X: P
AUSE 0: STOP
```

23627/8 VARS

Pointer to the start of the variables store. Apart from finding your way into the variables area, you can find the length of the BASIC program with this expression. This excludes screen, system variables, stacks and variables.

```
LET bytes=PEEK 23627+256*PEEK 23
628-PEEK 23635-256*PEEK 23636
```

23635/6 PROG

Address of start of the area in memory where the BASIC program is stored. This points to the first byte of the line number of the first program line. May be useful if you're converting programs for other Sinclair computers with information held as a REM statement in the first line of a program. See also under VARS above.

If you wish to "security-lock" a line into a program, then by means of this system variable you could POKE a zero into both bytes of a line number at the start of a program. Program lines start with a two byte line number.

23637/8 NXTLIN

Address of the start of the next program line. You could use this to enable you to access machine code stored within REM statements anywhere in the program, e.g. those loaded with MERGE from a tape library of subroutines. These would have their own local calls to machine code like this:

```
9000 LET A=USR (PEEK 23637+256*P
EEK 23638+5)
9010 REM <>MACHINE CODE<>
9020 RETURN
```

One constraint to this is that you should not include any colour, flash, brightness, etc., control characters into the REM statement or they may be interpreted as machine code,

upsetting things somewhat. However, if from a library of subroutines, these would not normally be used anyway.

23639/40 DATADD

This contains the address of the comma ending the last item of DATA. If nothing was read from the list (e.g. after RUN or restored, etc). the address held in 23639/40 is the address of the byte before the program area, normally the CHR$ 128 at the end of the channel information area. To demonstrate try RUNning this program:

```
  10 DATA "1","2","3","4","5"
  20 LET A=PEEK 23639+256*PEEK 2
3640
  30 PRINT A;TAB 9;PEEK A;TAB 16
;CHR$ PEEK A AND PEEK A>31
  40 READ B$
  50 GO TO 20


23754     128
23763     44      ,
23767     44      ,
23771     44      ,
23775     44      ,
23779     13
```

The address in this two byte system variable can point to the ENTER character or the colon signifying the end of the line or statement containing the DATA — the address of the terminator of the last item of data.

23641/2 E LINE

This system variable points to the start of the area above the variables. From this we can gain an idea of how much memory is used in bytes by screen, system variables, program and variables once the program has been RUN once to set up the variables, etc. Type this in, as a direct command:

```
PRINT PEEK 23641+256*PEEK 23642-
16384
```

We can also tell how much room is used for variables once the program has been RUN to set up the variables. Use the command:

```
PRINT PEEK 23641+256*PEEK 23642-
PEEK 23627-256*PEEK 23628
```

23653/4 STKEND

This system variable contains the address of where the spare part of memory starts. From reading this we can gain an idea of how much memory we have left by subtracting it from RAMTOP. This will not include memory used for the machine/GOSUB stacks but includes the length of the PEEK statement. So, this is only a fairly accurate guide but one which is adequate for most circumstances.

```
PRINT PEEK 23730+256*PEEK 23731-
PEEK 23653-256*PEEK 23654
```

23658 FLAGS 2

This system variable contains some flags used (normally) by the computer to indicate certain conditions.

The best use we can make of this is to make use of the flag indicated by bit 3. This being 1 indicates CAPS LOCK on or CAPS LOCK engaged.

What use is that? Consider in a program using INKEY$, e.g. in a menu of options in a filing program, we often need to know whether the operator is pressing a certain key. If the operator is invited to press Y for Yes or N for No he/she may press y for Yes or n for No — mix up lower case and upper case capitals. Most often this would depend on whether CAPS LOCK was engaged — people are not interested in upper or lower case — when they press y or Y they expect the computer to understand y as being Y like we would in everyday language. But the computer doesn't really appreciate that. So if we engage CAPS LOCK automatically, our worries are over and we have a simpler program which doesn't have to check (as far as it's concerned) two separate options for each choice.

It is tempting to use the BASIC statement POKE 23658,8 to engage CAPS LOCK and POKE 23658,Ø to disengage it. But this will affect the other flags, so do check their state first unless you know they are not any particular value. Normally in L mode, 23658 has a value of Ø so it is generally OK to use the POKEs above. You are not likely to cause crashes, but some

funny effects may rarely be caused. When the printer buffer is empty bit 1 will be zero.

23659 DF SZ

This system variable contains the number of lines in the lower section of the screen, normally used for INPUTs, error reports and so on. Normally this would be 2, except for when a long INPUT prompt is displayed, etc. If a value of 0 is POKEd in normally to attempt to clear this unused part so that we can use the whole 24 lines of the screen, the computer crashes. However, this can be done within a few restrictions. These restrictions are that we must ensure that the lower part of the screen is restored to normal before any use is made of this — so to BREAK out of a program would be somewhat catastrophic! Also, errors generated within the course of a program will have the same effect since the error report would have to be printed out. Here is a short listing to demonstrate the use of line 22 and 23 on the screen. Unfortunately, it only works for PRINT or PRINT TAB as we cannot use PLOT down here and PRINT AT will only work down to line 22. The screen is restored to normal by POKE 23659,2 *within* the program.

```
10 POKE 23659,0
20 FOR A=0 TO 23
30 PRINT A
40 NEXT A
50 PAUSE 0
60 POKE 23659,2
```

To demonstrate what can go wrong let us generate an error by adding this line to the program:

```
45 PRINT error
```

Oops!!! If you just want to PRINT on the bottom two lines it is usually better to use PRINT #1;"text" which works just as well if not better, without such a risk of causing a system crash. If you POKE a value greater than 2 into DF SZ the upper screen will become smaller than normal. So after POKE 23569,Y the upper screen would be 24 - Y rows down and would scroll when the PRINT position got to or beyond 24-Y,0. This program shows how a part screen scroll can be

maintained with DF SZ and SCR CT. Here random numbers appear and scroll up the top 14 lines of the screen only.

```
10 POKE 23692,0: POKE 23659,10
20 PRINT RND
30 GO TO 10
```

23670/1 SEED

When RANDOMIZE (number) is used, number (a constant or a variable) is stored in this system variable. This is the number that determines the next random number. This opens up the possibility of cheating since you could work out the next (supposedly) random number generated and use the knowledge gained to "swing" luck your way. For example, after RANDOMIZE 1 the next value of RND would be 0·0022735596 or INT (RND * 6) + 1 to simulate a die being thrown would be 1.

23672/3/4 FRAMES

This is a frame counter which can be used as a timer. It counts frames of a TV picture, so is incremented fifty times a second in the UK, or every 0·02 seconds, although the time taken to actually read and evaluate these three bytes of the timer may not allow it to be used to this accuracy. It has a timing range of nearly 4 days (actually about 3 days 21¾ hours). The manual (chapter 18) points out that you need to read the value of these three bytes twice in succession and take the higher value for full accuracy because of the possibility of the values of the three changing while being read in such a way as to cause large inaccuracies.

It must be emphasised that the timer bytes are in the opposite order to what you might expect — the most significant byte is 23674, so the timer values are ready by:

```
65536*PEEK 23674+256*PEEK 23673+
PEEK 23672
```

which returns time in units of fiftieths of a second. There are several things that affect the accuracy of this timer. Using BEEP stops the timer. Using the printer and loading/saving, etc., also affect its accuracy. However, the use of PAUSE is OK

as this only waits a specified time without resetting or stopping the timer.

23675/6 UDG

The address of the start of the dot patterns for the user defined graphics is normally 32600 on a 16K Spectrum or 65368 on a 48K Spectrum. This number is the same as USR "a", so PRINT USR "a" corresponds to:

```
PRINT PEEK 23675 + 256 * PEEK 23676
```

Compulsive POKEers can have fun with this one. The manual suggests changing this to save space by having fewer user defined graphics. However, it is also possible to do the reverse, set up more than one user defined graphics set if required although only one set of 21 can be in use at any one time. Remember that since there are 21 UDGs it is necessary to set aside 21 * 8 or 168 bytes for each separate set of UDGs and POKE the start addresses, into 23675/6, of the character set in use.

For fun, type in the following commands:

```
POKE 23675,96: POKE 23676,127 (16K Spectrum)
```

```
POKE 23675,96: POKE 23676,255 (48K Spectrum)
```

Then using the user defined graphics (they normally appear as capital letters until redefined) try to type out a message. I'll leave you to find out what happens.

One useful tip: once you've set up an user defined character set it may be SAVED on tape. Most people would type something like:

```
SAVE "chars" CODE 32600,168
```

Fine but you have to specify the start addresses. You could use SAVE "chars" CODE(PEEK 23675 + 256 * PEEK 23676),168 then you could happily save the current set of UDGs on tape without knowing the address of where they start. This would, for example, allow you to LOAD a character set SAVED from a 16K Spectrum back into a 48K Spectrum without having to know the addresses. To get a character set back into the right place on a machine with different amounts of memory simply

use LOAD "chars" CODE(PEEK 23675 + 256 * PEEK 23676),168. This would automatically relocate data to the right address for the machine in use at the time. This is the same as LOAD "chars" CODE USR "a" which saves a bit of typing although it may look a bit strange.

23679 P POSN

Contains information about how far across the printer buffer the LPRINT position has got to. Contains (33 - column number) for columns Ø to 31. You cannot change the LPRINT position by POKEing this alone.

23680 PRCC

Contains the low byte of the address where the next character is to go into the printer buffer, i.e. this will contain (23296 + LPRINT column number), being Ø for the left column of the printer, 15 for the 15th column, etc. Because this is the address of the top row of dots of each character you can POKE this to change the LPRINT buffer position provided you change the value in P POSN (23679) to match. It may appear to work if you don't do this but problems will be encountered at the end of the line.

23681 UNUSED SYSTEM VARIABLE

This system variable, although strictly speaking unused, usually contains 91. This is the high byte of the LPRINT buffer address (91 * 256 is 23296 where the buffer starts). This can be POKEd for your own use but using the printer will overwrite it back again to 91. 23680/1 together contain the address of the LPRINT position in the printer buffer. You will not affect the working of the printer if you POKE 23681 but anything stored here may be overwritten by the printer routines.

23677/8 COORDS

23677 is the system variable that contains the X co-ordinate of the last plotted point. After CLS it starts off at Ø and 23678 is the system variable that contains the Y co-ordinate of the last point plotted. It contains the actual value, so if the last point plotted was 3,3 both bytes would contain 3.

These two can be POKEd with valid X and Y co-ordinates respectively. Since POKEing these does not actually PLOT anything on the screen, this is a convenient way to move the PLOT cursor around. This could be done by PLOT OVER 1;X,Y:PLOT OVER 1;X,Y but would be messy. Amongst other things this could simulate MOVE found in other BASICs. Useful if you wanted to draw lines from around a particular point.

23684/5 DF CC

Address in display file of PRINT position. It may be POKEd to send the PRINT output elsewhere, although this requires an understanding of the way the display file is organised.

23688/9 SPOSN

23688 contains information concerning how far across the screen the PRINT position has got. It starts off as 33 for the left-hand side of the screen and decreases by one every time the PRINT position moves one place to the right. After using PRINT AT Y,X; (if Y and X are valid PRINT AT co-ordinates) 23688 would contain 33 - X. This can be useful when trying to prevent words being chopped in half when printed on the screen. If you imagine the number in 23688 as counting down towards zero as there is no more room on the current line, you can see that comparing this to the length of the word to be printed gives us an idea of whether it is necessary to move to a new line to prevent the word being chopped. Suppose the word to be printed was W$:

```
IF PEEK23688<LEN W$+1 THEN PRINT
```

This only works for words less than 32 characters long. 23689 contains information relating to how far down the screen the PRINT position has got to. It starts off at 24 for the top line of the screen and decreases by one every time the PRINT position moves down the screen. If you do not want a scrolling display and would rather the screen was cleared when the PRINT position got near the bottom of the screen, then try:

```
IF PEEK 23689=3 THEN CLS
```

23692 SCR CT

Contains how many scrolls will be carried out + 1 before waiting with scroll? to give viewing time. In graphics games

especially this can be a nuisance, since one is not interested in waiting for viewing. So if the number in 23692 is anything other than 1 the waiting does not occur. So POKE 23692,255 would give you 255 lines of printing before the machine waits with scroll?. POKE 23692,Ø seems to have a similar effect except that you have one more line of print. If more is needed, then if the printing is done within a loop it will be necessary to include the POKE statement in the loop as well — wasting time but necessary.

23693 ATTR P

Contains permanent attributes, or the attributes (FLASH, BRIGHT, PAPER, INK) in effect *globally*. Local colours in PRINT statements, etc., are dealt with elsewhere. Note that most of the ROM routines use the values of the system variable holding the *temporary attributes* as these contain the permanent attributes unless a local parameter is specified. CLS, however, clears the screen to the colours, etc., in ATTR P. The functions of the individual bits are as follows:

bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit Ø
FLASH	BRIGHT	PAPER colour in binary			INK colour in binary		

Bit 7 is 1 for FLASH 1.

BIT 7 is Ø for FLASH Ø.

BIT 6 is 1 for BRIGHT 1.

BIT 6 is Ø for BRIGHT Ø.

Bits 5,4 and 3 contain the PAPER colour in binary, e.g. for

PAPER 7 bits 5, 4 and 3 would be 111.

Bits 2,1 and Ø contain the INK colour in binary, e.g. for

INK 3, bits 2,1 and Ø would be Ø11.

Attributes of 8 or 9 are not dealt with here. If the permanent attributes are 8 or 9 then these stored in 23693 may not be valid.

23694 MASK P

This is the system variable that helps the Spectrum determine the attributes of anything printed when a parameter of 8 is specified. So if you specified BRIGHT 8 globally, bit 6 of 23694 would be set to 1 to remind the computer in future that BRIGHT 8 has been specified. So to determine the colour/flashing/brightness when printing, the computer looks at what's already there and prints the word in that colour, etc. Or if you like, it only overprints the character on the screen and leaves the attributes alone. This is what each bit does:

bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
FLASH 8	BRIGHT 8	PAPER colour 8?			INK colour 8?		

Bit 7 is 1 when FLASH 8 is in effect.

Bit 6 is 1 when BRIGHT 8 is in effect.

Bit 5,4 and 3 are normally all 1 when PAPER 8 in effect, but see below.

Bit 2,1 and 0 are normally all 1 when INK 8 in effect, but see below.

Where there is more than one bit to consider, as in INK and PAPER, then only the bits set have their attributes bit taken from the screen. This can lead to some unexpected effects. Try this:

```
10 INK 8
20 POKE 23694,BIN 00000011
30 PRINT AT 0,0; INK 5;"555555
5"
40 PRINT AT 0,0;"1111"
```

As INK 8 is specified, you may expect the ones to be printed in cyan like the fives, but no. Rather than check the INK attribute as a whole, it only checks the bits set in 23694, which were bits 0 and 1. See if you can work out what colour the ones will be printed in. Have fun!

23695 ATTR T

This system variable contains the current temporary colours as would be set up by local statements within PRINT statements.

You could see this for yourself with something like these two direct commands:

PRINT PEEK 23695

PRINT INK 7;PAPER Ø;PEEK 23695

That is, include the PEEK in a PRINT statement under the effect of the local colour controls. Normally, unless local colour statements are specified, this system variable will contain the global colour values. Colours, etc., to be used for printing on screen are taken from these temporary system variables and things are balanced such that ATTR T is only different from ATTR P if local colour attributes and so on so decree. This is the function of individual bits:

bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit Ø
tempor-ary FLASH	tempor-ary BRIGHT	temporary PAPER colour			temporary INK colour		

23696 MASK T

This is rather like MASK P (system variable 23694) except that the parameters in here are temporary. Normally the same as the equivalent permanent parameter 8s, this is changed while local colour 8s, etc., are in effect. You could study this by using something like:

```
PRINT PEEK 23696, INK 8;PEEK 236
96, INK Ø; FLASH 8;PEEK 23696
```

The individual bits have the following functions:

bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit Ø
temporary FLASH 8?	temporary BRIGHT 8?	temporary PAPER 8?			temporary INK 8?		

23697 P FLAG

This system variable contains, as you might expect from its name, flags used during printing. After PAPER 9 has been specified, bits 6 and 7 are set to 1. After INK 9 has been specified, bits 4 and 5 are set to 1. After INVERSE 1 has been specified, bits 2 and 3 are set to 1. And after OVER 1 has been specified, bits Ø and 1 are set to 1. The effects are global if the odd num-

bered bits (bits 1,3,5 and 7) are set to 1 and temporary if the even bits (bits 0,2,4 and 6) are set to 1, as this diagram shows:

bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
global PAPER 9	temporary PAPER 9	global INK 9	temporary INK 9	global INVERSE 1	temporary INVERSE 1	global OVER 1	temporary OVER 1

23681,23728/9

These three bytes in the system variables are not normally used by the Spectrum — you may like to make use of them as "custom variables" for use in your own programs in which to access information. These are particularly useful in machine code routines where you can simply access the information by an address rather than searching for the variable in the variables area. 23728/9 was intended for use by non-maskable interrupts but these don't occur on a bare Spectrum.

23730/1 RAMTOP

This two byte system variable points to the last byte of RAM of the BASIC system area. Note that this is not the end of the memory used by BASIC, in the sense that the user defined graphics normally hide up above this address. If you move RAMTOP up above the start of the user defined graphics they may be overwritten, but you gain quite a few valuable bytes which may be useful for 16K users.

One important thing is that NEW only operates as far as the address held in 23730/1 so you can store data above this which may be passed between programs loaded into the computer. The same is true if you want to preserve machine code routines, etc.

23732/3 P RAMT

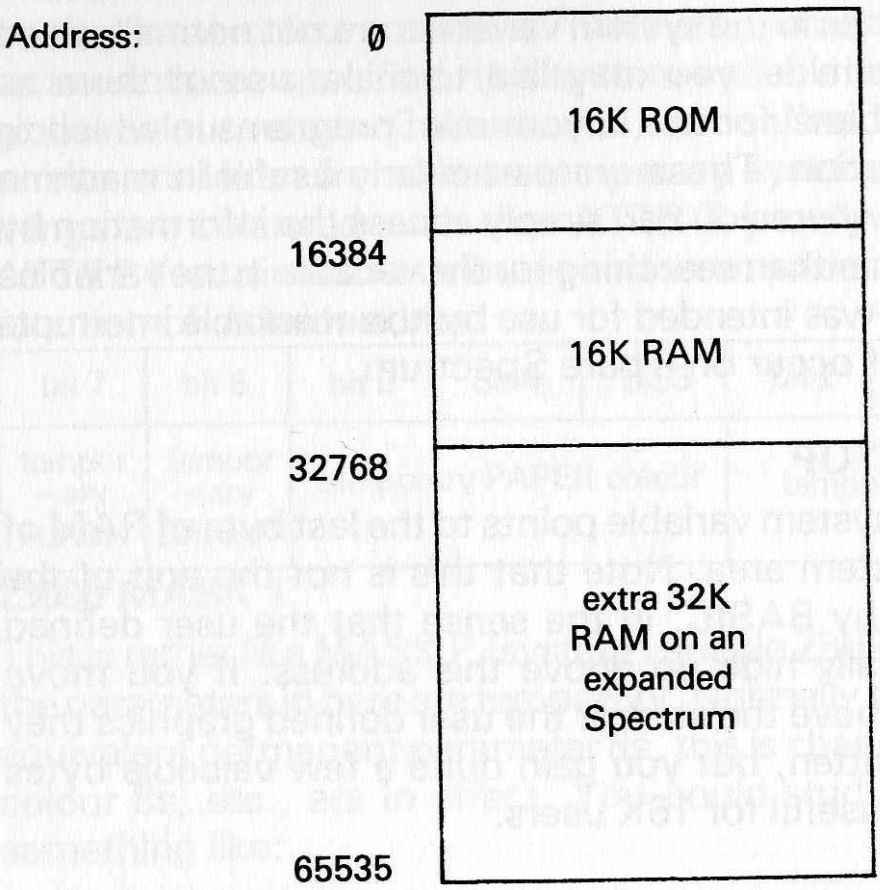
This contains the address of where RAM ends on the Spectrum. If you acquire a Spectrum whose memory capacity you don't know, then you do not need to look inside it to see if it's an expanded model or not, just enter this expression:

```
PRINT PEEK 23732 + 256 * PEEK 23733 - 16384
```

The 16384 bytes subtracted is for the ROM since RAM starts at address 16384 and goes up to the address held in P RAMT.

THE LAYOUT OF THE SPECTRUM'S MEMORY

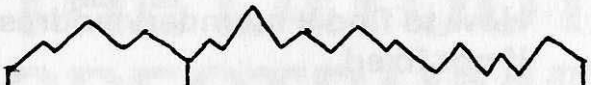
The Spectrum normally consists of 16K of Read Only Memory (ROM) space and either 16K RAM (Random Access Memory) or 48K RAM, as shown in this diagram:



The 16K ROM contains the instructions, data and so on that the Spectrum needs to run programs, execute instructions supplied by the user, drive the printer, tape recorder and so on. This area consists of 16384 bytes (0 to 16383). After this comes another 16384 bytes of RAM and another 32768 bytes of RAM on the expanded 48K Spectrum.

The main difference between ROM and RAM is that the contents of ROM are fixed and cannot be altered and so remain even if the power is switched off. RAM contents can be altered quite easily, almost at will, and, of course, the contents are lost when the power is turned off. The RAM is divided into many sections as in this diagram:

Address	Area Of Memory	How to find a boundary address if not fixed
16384	Display file	
22528	Display attributes	
23296	Printer buffer	
23552	System variables	
23734	Microdrive maps (if microdrives are connected)	
CHANS	Channel information	PEEK 23631 +256*PEEK 23632
	CHR$ 128	
PROG	The BASIC program	PEEK 23635 +256*PEEK 23636
VARS	Variables store area	PEEK 23627 +256*PEEK 23628
	CHR$ 128	
E LINE	Command or line being edited	PEEK 23641 +256*PEEK 23642



	Command or line being edited	
	CHR$ 13	
	CHR$ 128	
WORKSP	INPUT data	PEEK 23649 +256*PEEK 23650
	CHR$ 13	
	Temporary workspace follows after the CHR$ 13 marker	
STKBOT	The calculator stack	PEEK 23651 +256*PEEK 23652
STKEND	Spare memory	PEEK 23653 +256*PEEK 23654
Z80A stack pointer	Machine stack used by the Z80A microprocessor	(not accessible from BASIC)
	GOSUB stack	
RAMTOP	CHR$ 62	PEEK 23730 +256*PEEK 23731
UDG	User defined graphics	PEEK 23675 +256*PEEK 23676
P-RAMT		PEEK 23732 +256*PEEK 23733

Let us now take a brief look at these sections one by one where they are of interest.

(1) Display file Address 16384 to 22527

This is the copy of the television screen picture that resides in the computer's memory. For every dot on the screen in both the upper part of the screen and the lower part of the screen (256 * 192) there is a matching bit in this area. It consists of 6144 bytes which are laid out rather curiously at first sight as the manual explains. If a point, or dot, on the screen is set to an INK colour then the relevant bit is set to 1 in the display file. This brief program will give you an idea of the layout of the display file:

```
10 FOR A=16384 TO 22527
20 POKE A,255
30 NEXT A
```

(2) Attributes file Address 22528 to 23295

This area of memory contains information about the colours, brightness and flashing effects on the screen. If you consider one character space on the screen as being an 8 by 8 grid of dots then the dots within that grid can only have one INK and one PAPER colour because all their attributes come from the same place — the attributes operate on the same grid as the PRINT routine. That's why you can't have a green spaceman with magenta eyes on a black background or whatever. One byte of eight bits in the attributes area contains the information for FLASH, BRIGHT, PAPER and INK for one character location on the screen. There are 768 attribute bytes (32*24) including the bottom two lines of the screen used for INPUTs, etc. These are stored in the order: first line of 32 bytes for the first screen line of thirty two characters, second line of 32 attribute bytes for the second line of 32 characters across and so on. This short program will demonstrate this:

```
10 FOR A=22528 TO 23295
20 POKE A,199
30 NEXT A
```

Three bits are used for PAPER, three bits used for INK, one bit for FLASH, and one bit for BRIGHT. They are all stored in binary, so one bit alone gives 0 or 1 and three bits give 0 to 7. You may be aware that parameters of 8 or 9 (e.g. FLASH 8 or

PAPER 9) are not stored here — only the *result* after the computer has decided if INK 9 will end up red or white or whatever. This is what each bit signifies in an attributes byte:

bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
FLASH	BRIGHT	PAPER colour			INK colour		

(3) Printer buffer Address 23296 to 23551

Information about characters waiting to be sent to the printer. It consists of 256 bytes or 32 by 8 so that graphics can be sent to the printer. The actual dot pattern as will appear on the printer paper is stored here.

(4) The system variables Address 23552 to 23734

Bytes containing information relating to several factors about the computer's operation — where various sections of memory such as the user defined graphics, variables, etc., start and end. They all have a name, but the computer will not recognise this since it is only for the programmer's benefit. These names are SCR_CT, VARS and so on.

(5) Microdrive maps

This area contains information relating to the microdrives when they are connected. If the microdrives are not connected, there is nothing there and the next system variable pointer CHANS points to 23734. No further information was available at the time of writing this book.

(6) Channel information

On a standard 16K or 48K Spectrum with nothing attached (except possibly a printer which makes no difference to the contents of this area), there are details of 4 INPUT/OUTPUT channels in this area. These refer to what information comes from where and goes to where, e.g. if anything typed at the keyboard appears on the lower part of the screen, and the addresses of the routines that handle the INPUT/OUTPUT operations for that channel.

The area starts at the address stored in the system variable CHANS 23631/2 and ends at the address held in the system variable PROG 23635/6 less one with a byte holding CHR$ 128 which signifies the end of the channel information area. In terms of bytes on the 16K or 48K bare Spectrum (irrespective of whether or not the printer is connected) the area is 21 bytes long from 23734 to 23754. The numbers held in this area are as follows. They are arranged in a table of 4 by 5 figures:

```
244    9    246   16   75   [K]
244    9    246   21   83   [S]
129   15    131   21   82   [R]
244    9    246   21   80   [P]
128
```

The characters on the right are the "file names" of the four channels, so we have channel K, channel S, channel R and channel P.

Channel K is the "Keyboard" channel. Information can come from the keyboard and go to the lower part of the display. Channel R allows information to be sent to the work space area and channel P allows information to be sent to the printer. So what are the four bytes before the file names? Let's rewrite this as two addresses and file names, the addresses being those of the routines handling the appropriate routines:

(decimal) output routine address	(decimal) input routine address	filenames
2548	4254	K
2548	5572	S
3969	5572	R
2548	5572	P

(7) Program area

This is the area where the BASIC program goes. The system variable PROG points to the start of the program area, to the first byte of the first line number. This follows a byte with CHR$ 128 marking the end of the channel information area. This area ends at the address held in VARS less 1. The lines of a BASIC program are stored like this:

Two bytes holding line number MSB | LSB	Two bytes holding length of text of line + 1 for the CHR$ 13 at the end of the line	Text of the program line	CHR$ 13 BIN 00001101 (ENTER)

Each line ends with a CHR$ 13 or the ENTER character. Multiple statements are separated by a colon, CHR$ 58, and numbers are stored twice, firstly as the CODEs of their digits, then after a CHR$ 14 (indicating a number) follows a five byte representation of the number be it integer format or floating point.

This simple program will allow you to examine any line of BASIC you place as the first line of a program. It works out where the first program line starts (line 1Ø) and where it ends (line 2Ø) by using the bytes three and four in the program line, which hold the length of the line from byte 5 of the program line to the ENTER character at the end. All addresses are PEEKed in turn and three columns printed on the screen. The first shows the addresses, the second the number in that memory location and the third the character corresponding to that number if this is a printable character. The example shows a REM statement being examined:

```
23755    0
23756    1
23757    15
23758    0
23759    234      REM
23760    108      l
23761    105      i
23762    110      n
23763    101      e
23764    32
23765    111      o
23766    102      f
23767    32
23768    66       B
23769    65       A
23770    83       S
23771    73       I
23772    67       C
23773    13
```

```
   1 REM line of BASIC
  10 LET START=PEEK 23635+256*PE
EK 23636
  20 LET FINISH=START+3+PEEK (ST
ART+2)+256*PEEK (START+3)
  30 FOR A=START TO FINISH
```

```
40 LET CHAR=PEEK A
50 PRINT A;TAB 8;CHAR;TAB 16;C
HR$ (CHAR) AND CHAR>31
60 NEXT A
```

(8) Variables area

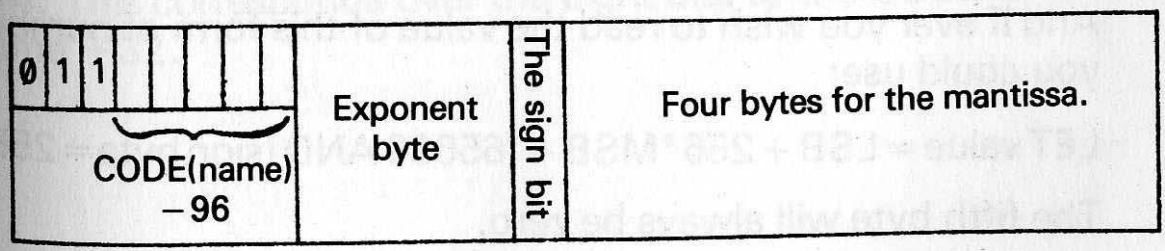
The variables area starts at the address held in the system variable VARS and ends at the address held in system variable E-LINE. This is where the variables used by BASIC are stored. To avoid confusion the variable names are stored with different formats to their names for the names of strings, arrays, numeric variables, etc.

Different patterns of bits for the first letter in the name of the variable enable the computer to tell the difference between the different types.

Usually these letters are stored as lower case letters but the different bit patterns mean that we can not always look for the CODE of the letter in the name, for example. We will now take a brief look at the different types of variables. The first letter of all variable names is stored in one byte as the CODE of the letter less 96, since only bits 0 to 4 are significant for the name. Bits 5, 6 and 7 always have values that depend on the type of variable so are not used in the name. The values, if bit 5 and 6 were 1, would be 32 and 64 respectively, the sum of which is 96. The actual letter is stored in bits 0 to 4 as the CODE of the lower case letter minus the value of these two bits. So if the first letter of the variable name was an "a", bits 4 to 0 would be BIN 00001. Bits 5, 6 and 7 would depend on the type of variable.

Variables whose name consists of one letter (numeric variable)

This type of variable is stored with bits 5 and 6 of its name set to 1 and bit 7 reset to 0. So this is a rare example of the first character in the name being stored as its name exactly, but only because the bit patterns happen to coincide. The name is followed by the exponent byte and four bytes for the mantissa:



The exponent byte has a value of 1 to 255 (may be 1 or 255). This shows where the decimal point lies. The mantissa is held in four bytes. These give the digits of the number and can have a value of Ø·5 to 1 (it is never 1, although it can be Ø·5). The number can be found with the formula (mantissa)*2↑(exponent – 128).

As a result of the values which the mantissa can take, the left-most bit of the mantissa is used as a sign bit. This is 1 if the number is negative or Ø if the number is positive.

There is a slightly different method of storing whole numbers from –65535 to +65535, numbers which may be accommodated in two bytes:

<table>
<tr><th>byte 1</th><th>byte 2</th><th>byte 3</th><th>byte 4</th><th>byte 5</th></tr>
<tr><td>Ø
zero</td><td>Ø if number positive or 255 if negative</td><td colspan="2">LSB | MSB
The number is held in bytes three and four</td><td>Ø
zero</td></tr>
</table>

The first byte of numbers held in this way is always zero. This could aid identification of this format of numbers. The second byte is the sign byte — it will be Ø if the number is positive or it will be 255 if the number is negative. The third and fourth bytes contain the value of the whole number in the order Less Significant Byte (LSB) followed by the More Significant Byte (MSB) and is held in what is called "twos complement form".

The value for a positive number can be read by the expression:

LSB + 256*MSB

The value for a negative number can be found by the expression:

LSB + 256*MSB – 65536

And if ever you wish to read the value of this form of number, you could use:

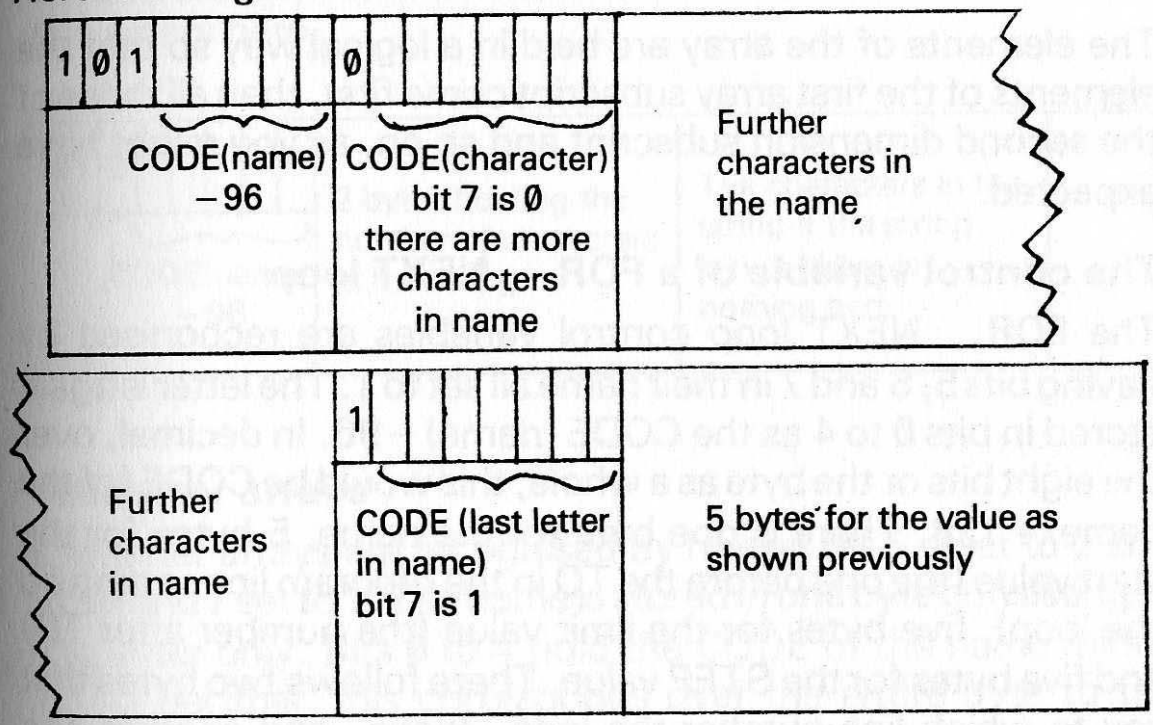
LET value = LSB + 256*MSB – (65536 AND (sign byte = 255))

The fifth byte will always be zero.

Numeric variables whose names consist of more than one letter

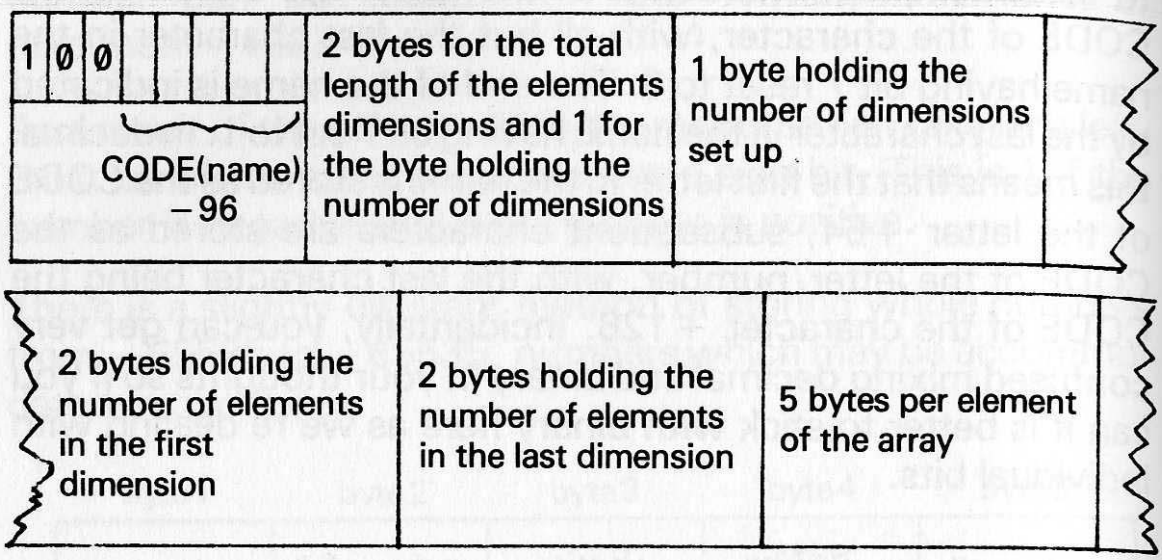
This type of variable is stored with bits 5 and 7 of the first letter in the name set to 1 and bit 6 of the first letter in the name reset to Ø. Subsequent characters in the name are stored as the CODE of the character, with all but the last character in the name having bit 7 reset to Ø. The end of the name is indicated by the last character in the name having bit 7 set to 1. In decimal this means that the first letter in the name is stored as the CODE of the letter +64, subsequent characters are stored as the CODE of the letter/number, with the last character being the CODE of the character +128. Incidentally, you can get very confused mixing decimal and binary in your thoughts so if you can it is better to stick with binary here as we're dealing with individual bits.

Here is a diagram to illustrate this:



Array of numbers

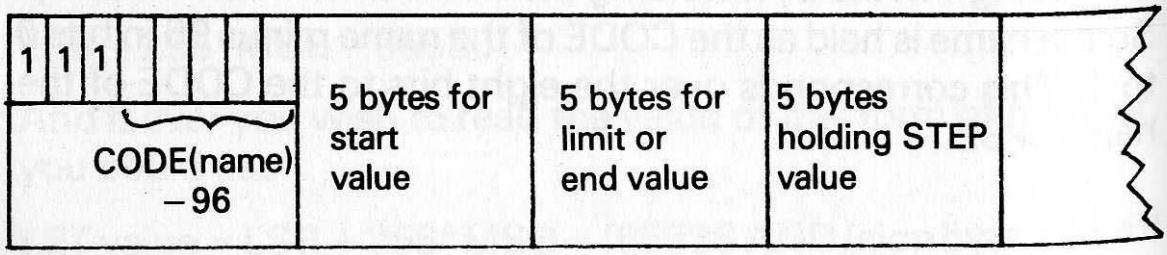
An array of numbers is recognised by bits 5 and 6 being Ø and bit 7 being 1 in the byte holding the name of the numeric array. So the name is held as the CODE of the name minus 96 in bits Ø to 4. This corresponds over the eight bits to the CODE of the name +32.

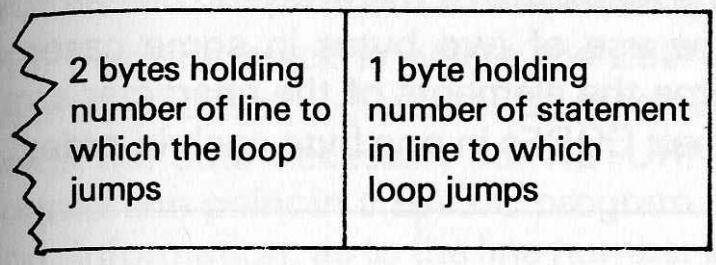


The elements of the array are held in a logical way so that the elements of the first array subscript come first, then all those of the second dimension subscript and so on, as you might have expected.

The control variable of a FOR...NEXT loop

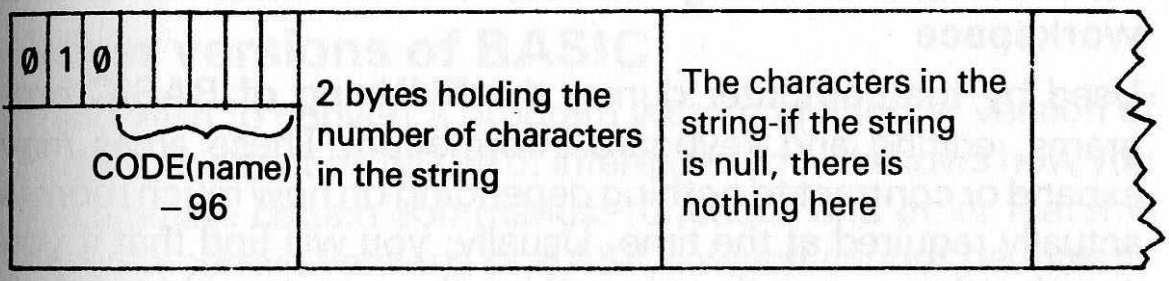
The FOR...NEXT loop control variables are recognised by having bits 5, 6 and 7 in their name all set to 1. The letter is again stored in bits 0 to 4 as the CODE (name) – 96. In decimal, over the eight bits or the byte as a whole, this would be CODE (of the name) + 128. There is one byte for the name, 5 bytes for the start value (the one before the TO in the program line setting up the loop), five bytes for the limit value (the number after TO) and five bytes for the STEP value. There follows two bytes that say to which line number the loop "loops" and another one byte for the number of the statement within that line to which the loop jumps to. This makes a total of 19 bytes occupied in the variables area.





String variables

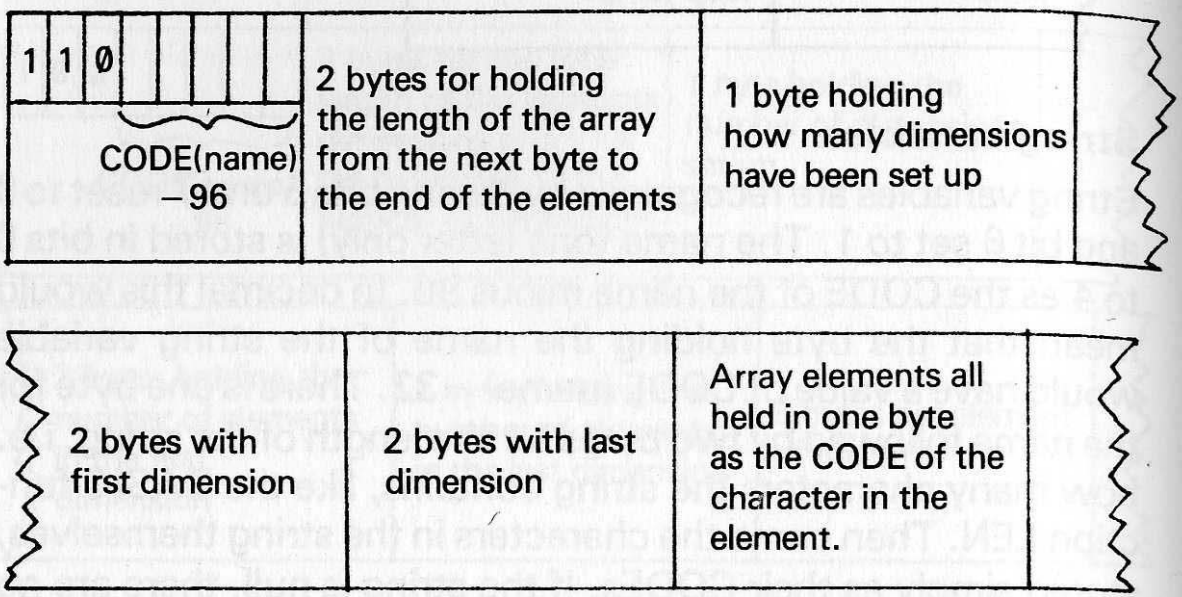
String variables are recognised by having bits 5 and 7 reset to Ø and bit 6 set to 1. The name (one letter only) is stored in bits Ø to 4 as the CODE of the name minus 96. In decimal this would mean that the byte holding the name of the string variable would have a value of CODE (name) – 32. There is one byte for the name followed by two bytes for the length of the string, i.e. how many characters the string contains, like the BASIC function LEN. Then come the characters in the string themselves, stored simply as their CODEs. If the string is null, there are no characters here.



Character arrays

Character arrays are recognised by having bit 5 reset to Ø and bits 6 and 7 set to 1. The name is stored in one byte consisting of one letter only. Bits Ø to 4 hold the CODE of the name minus 96. In decimal, this corresponds over the entire byte to the CODE of the name + 96. There is one byte for the name then two bytes holding the length of the remainder of the string including the elements, the dimensions and an extra one for how many dimensions there are. The length bytes are followed by one byte holding the number of dimensions set up. If you had DIM A$(4,5,6) this byte would hold 3. The dimensions themselves follow in two bytes each in the order written down in the program line setting up the DIM. So in the example of DIM A$(4,5,6) you would get, after the 193 for the name array, the numbers 127,Ø,3,4,Ø,5,Ø,6,Ø followed by the CODEs

of the characters in the elements of the arrays. The zeros, of course, are due to the use of two bytes in some cases as described. Finally, come the elements of the character array. These are stored by their CODEs in one byte each in order.



Command/line being edited, INPUT data and the workspace

Used by the computer during the RUNning of BASIC programs, editing and keyboard information. These areas may expand or contract to nothing depending on how much room is actually required at the time. Usually, you will find that if you examine these areas there is nothing there as everything is balanced so that any space not required is not used.

The calculator stack

The calculator stack is used to place floating point numbers, integers and information about strings. Everything generally goes on as five bytes — numbers in their five byte form and strings as five bytes of details about them (starting addresses and so on). Arithmetic operations and so on make use of this stack.

Spare memory

This is the part of memory between the calculator stack and the machine stack which is not required. It may come into use if stacks grow or changes are made to variables or program, etc. It is not safe to put anything of importance here.

Machine stack and GOSUB stack

The machine stack is where the Z80A may put information such as values of registers to be preserved. The GOSUB stack holds the data necessary for RETURN to jump back to the appropriate point in a BASIC program after a BASIC subroutine. Information, as to the line number and statement number after the GOSUB, is stored here which is used to know where to RETURN to.

RAMTOP

This is the limit of memory used by BASIC, apart from the user defined graphics data which is stored between RAMTOP and the end of available memory. Anything stored above RAMTOP in RAM is safe from everything in BASIC except POKE. NEW only operates as far as the address held in RAMTOP so UDGs set up are safe unless you turn off the machine or POKE over them. The same applies to any data or machine code above RAMTOP — it is even safe from NEW.

Other versions of BASIC

If you wish to convert a program written in another version of BASIC, this section will be of interest to you. It shows how you can emulate certain commands, functions and other features found in certain versions of BASICs which cannot be directly used in Sinclair Spectrum BASIC. Sometimes, it is just a matter of using another word or sometimes a complete routine may be necessary.

Arrays

With numeric arrays, the only problem likely to arise is with subscripts. These may start at 0 whereas Spectrum subscripts start with 1. The solution generally is to add 1 to all references to any array subscripts, including DIM statements, if the program makes use of subscript 0. Watch out for calculated references, e.g. A(G*3 – 1) which can be tricky to convert — you will have to convert these individually as you meet them.

ASC

This corresponds to the Spectrum's CODE function — at least, for characters with CODEs of 32 to 126 on the Spectrum. If this

is used for storing DATA in a string or array and subsequent decoding, you may be all right not having to do any changes – otherwise study the routine to find out whether any values or characters involved differ from their equivalent on the Spectrum. Any changes will have to be made individually for each program as there is no hard and fast rule as to what characters have what CODEs/ASC values outside the range 32 to 126. There may even be one or two differences within this range.

CALL

This is used to jump into machine code and may be replaced by USR. The Spectrum returns a number when coming back into BASIC which you'll have to find something to do with, e.g., CALL address could be replaced with LET A = USR address or RANDOMIZE USR address.

Degrees and Radians

The Spectrum's trigonometric functions work in radians. If you need to adapt a routine using degrees they may be changed to radians with:

LET RADIANS = (PI*DEGREES)/180

DIM

The program may call for several arrays to be set up using one DIM statement all separated by commas. So you would replace DIM A(3),B(4),C(5) with DIM A(3): DIM B(4): DIM C(5). Beware of DIM statements setting up string arrays or character arrays, because you may need an extra subscript on the Spectrum since these arrays are all set to a fixed length at the time of the DIM statement. Normally on other computers, the length of each string in the array is only as long as it needs to be. So, if the program calls for DIM A$(4) this means four separate strings, not the one string of four characters as this statement might appear to ZX users. The way to convert this for the Spectrum is to work out what the longest string is, say 10, letters and set up the Spectrum DIM accordingly as DIM A$(4,10).

DIV

This function returns the whole number part after a division so that 10 DIV 4 would be 2. Rewrite this as INT(10/4), i.e. put the division in brackets and apply INT to this.

DO...UNTIL

The program carries on executing the part of the program between DO and UNTIL and only stops doing this once the condition after UNTIL has happened. Normally, you'd convert this with IF...THEN GOTO...

DRAW

Most BASICs would draw a line by specifying the end co-ordinates, e.g. DRAW 200,90 would draw a line ending at pixels 200 across and 90 vertically. This would cause problems because the Spectrum's DRAW routine would expect relative x and y values, i.e. x pixels across from where the line started and y pixels up from where the line started. Note that most BASICs would MOVE the PLOT cursor first then DRAW from this. So to convert MOVE a,b:DRAW x,y, but for the Spectrum you'd use PLOT a,b:DRAW (x−a),(y−b). Note that there is a difference in that the PLOT has to be used both for MOVE and to shade in the first dot in the line.

ELSE

This is a part of IF...THEN...ELSE in which the condition after IF being false the rest of the line is not skipped over but that action after ELSE is done instead. You will have to write out two separate courses of action. IF X = 1 THEN PRINT "X = 1" ELSE PRINT "X is not 1". This would be written as:

IF X = 1 THEN PRINT "X = 1"

IF X<>1 THEN PRINT "X is not 1"

You may also be able to rewrite some examples using AND to join the conditions. However, using two IF...THEN statements is the best course to follow normally.

END

Under some circumstances this may simply be omitted. It is to all intents and purposes the same as STOP in most cases.

Exponentiation

Different BASICs use different symbols for this — just replace the ** or ∧ with the Spectrum's ↑.

FOR...NEXT loops

The difference you are likely to encounter is where the program tests to see if the loop has been finished. If the test is done at the NEXT end of the loop, this means that the loop will be run through at least once. On the Spectrum, the test is done at the FOR end of the loop, meaning that the loop is never executed if the finish value is already exceeded by the start value being bigger than the end value.

GET and GET$

This reads the keyboard and usually waits for a key to be pressed before the program moves on. The full expression is likely to be LET A = GET or GET A (and the same for GET$). This can be converted using INKEY$ (if the computer concerned waited for a key to be pressed, you'd made special arrangements) or using INPUT (you'd have to make it obvious ENTER was needed).

If you use INKEY$ convert LET A = GET or GET A with:

```
1000 LET A = CODE INKEY$: IF A = 0 THEN GOTO 1000
```

With GET$ (e.g. LET A$ = GET$):

```
1000 LET A$ = INKEY$: IF A$ = ""THEN GOTO 1000
```

IF...THEN...

Some BASICs allow THEN to be omitted. However, it must be included on the Spectrum. For example, IF X = 1 PRINT "One" must be written as IF X = 1 THEN PRINT "One"

INKEY

This reads the CODE of the key just pressed on the keyboard. It can be converted by CODE INKEY$ so that LET A = INKEY would become LET A = CODE INKEY$.

You may come across a version with a number in brackets after it. This is the time the function will wait for a key to

be pressed before giving up and moving on. The easiest way is to put a PAUSE before the keyboard scan. So:

LET A = INKEY(50)
may become:

PAUSE 50: LET A = CODE INKEY$

If the time units are not the same, some change to the length of the PAUSE may be necessary.

INSTR

This function looks for a copy of one string within another. You will need a complete routine to replace this. The routine shown will produce similar results to LET Y = INSTR(B$,C$) although it will be a great deal slower. B$ is the "big" string in which you look for a copy of the smaller string C$. No errors occur if either string is biggest or both the same size.

```
9200 REM INSTR
9205 LET P=0
9210 IF LEN C$=0 OR LEN B$=0 OR
LEN C$>LEN B$ THEN RETURN
9215 FOR P=1 TO LEN B$-LEN C$+1
9220 IF B$(P TO P+LEN C$-1)=C$ T
HEN RETURN
9225 NEXT P
9230 LET P=0
9235 RETURN
```

On returning from the routine, Y will contain the position of where the copy starts — 1 if the copy starts at the beginning, 2 if it starts at the second letter and so on. If no copy is found or C$ is bigger than B$ then Y will be 0. Whatever the reason, Y being 0 means there is no exact copy of C$ in B$.

Integer variables

These are normally identified by having a % symbol tagged onto the end of the name. For example, A%. Normally, you could use any variable name you like as long as you made sure the value going in the variable was an integer, e.g. if the original read LET A% = 3/2 and you want to PRINT A%, you would get 1. To have the same result on the Spectrum you'd use LET A = INT(3/2).

LEFT$

The function LEFT$(A$,B) which takes the first B characters from A$ may be written as A$(TO B) on the Spectrum. So something like LET R$ = LEFT$(A$,B) would become LET R$ = A$(TO B).

LINK

This is a call to a machine code routine. You may replace it with USR.

Logical expressions

You are likely to encounter problems with TRUE and FALSE values. Zero is always the false value in programs you will encounter, but true may often be −1. By this I mean that something like PRINT (1 = 2) will print Ø (as it would on the Spectrum) and PRINT (1 = 1) will print −1 because the expression is true. On the Spectrum, PRINT (1 = 1) would print 1. The solution here is to negate the result of these expressions. If the original read:

```
LET X = 1Ø − (SCORE = 6) + (TIME < 3ØØ)
```

this would be rewritten for the Spectrum as:

```
LET X = 1Ø + (SCORE = 6) − (TIME < 3ØØ)
```

Beware of uses of AND and OR as binary operators which operate on numbers bit by bit. These usually manifest themselves in this sort of statement:

```
LET A = A AND 4
```

There is no easy way to convert these in BASIC. You will have to rewrite the routine generally or forget it completely.

LOGS

The Spectrum uses natural logarithms. If you require logs to other bases (usually base 1Ø) use:

```
LET LOGBASEX number = LN number / LN X
```

where X is the base and number is the number you wish to find the logarithm of.

MAT

Short for Matrix or Matrices. This is a function which will work on all elements of an array:

```
10 DIM X(10)
20 DIM Y(10)
30 MAT X = Y
```

Thus, all the elements of the array X are made equal to the elements of array Y. You will have to use a loop to perform the operation on all elements of the arrays:

```
10 DIM X(10)
20 DIM Y(10)
25 FOR M = 1 TO 10
30 LET X(M) = Y(M)
35 NEXT M
```

MID$

MID$(W$,T,U) takes the middle U characters of W$ starting at element T. On the Spectrum use W$(T TO T + U − 1).

MOD

A MOD B gives the remainder after A has been divided by B. Replace with A − (INT(A/B) * B).

MOVE

All this does is alter the position of the PLOT cursor without plotting anything on the screen. It is generally used to set up a point from which a line is drawn, and for examining a particular part of the screen. Note that when used for drawing lines, the associated function DRAW would fill in the first dot in the line — the Spectrum DRAW would not and you would use PLOT and DRAW together. To simulate MOVE by itself you have to POKE the X and Y values into the system variables holding the current PLOT co-ordinates. To replace MOVE on the same scale on the screen (e.g. MOVE X,Y) use:

```
POKE 23677,X: POKE 23678,Y
```

NEXT

In some versions of BASIC, the name of the variable after the NEXT may be omitted. If so, the value of the most recent con-

trol variable is incremented. This is not possible on the Spectrum. You must always specify the control variable's name.

ON...GOTO/GOSUB...

This usually takes the form:

```
ON X GOTO 200,300,400,500
```

Which means that if X was one, the program would GOTO the first line number in the list after the GOTO or GOSUB. If X was 2, the program would GOTO/GOSUB the second line number and so on.

The easiest way is to convert using IF...THEN GOTO... so the above would become:

```
IF X = 1 THEN GOTO 200
IF X = 2 THEN GOTO 300
IF X = 3 THEN GOTO 400
IF X = 4 THEN GOTO 500
```

You could use AND to convert it:

```
GOTO (200 AND X = 1) + (300 AND X = 2) + (400 AND X = 3) + (500 AND X = 4)
```

If the line numbers go in a step suitable to allow it, use a computed GOTO like this:

```
GOTO 100 + (X * 100)
```

PAINT

This is a graphics command which fills in an area of the screen in colour. There is no fast way to convert this in BASIC and anyway it tends to vary from machine to machine. This and many other graphics commands are best avoided because graphics is probably the area where the biggest differences between various computers are found.

PEEK and POKE

This is the other big difference from computer to computer. A specialist knowledge of the computer concerned is generally needed and is often impossible to convert.

PRINT

On some computers the question mark ? is used as an abbreviation for the command PRINT, which must be used on the Spectrum.

PROC,ENDPROC

PROC is short for procedure, which is a form of subroutine called by name rather than line number. On the Spectrum you would replace the PROC with a subroutine, called by GOSUB (line number). Replace the ENDPROC with a RETURN.

RANDOM NUMBERS

Some computers generate random numbers with the expression RND (X) which returns a whole number between 1 and X inclusive. Convert RND (X) with INT(RND *X) + 1. If you see an expression like RND (Ø) it usually means repeat the last random number. RND (– X) usually means the same as RAND on Spectrum.

REPEAT...UNTIL

This is a means of creating a loop without reference to line numbers. The loop keeps going forever, unless the condition mentioned after the word UNTIL becomes true and the program finishes the loop and carries straight on. Here is an example of its use:

```
10 LET X=0
20 REPEAT
30 LET X=X+1
40 UNTIL X=10
```

This can be converted using an IF...THEN GOTO... statement in the line containing the UNTIL, the GOTO referring to the statement or line after the one containing the REPEAT. If you convert it as I have done here, you may as well include a REM statement as a reminder and GOTO this:

```
10 LET X=0
20 REM loop here
30 LET X=X+1
40 IF X<10 THEN GO TO 20
```

RESET

This is used to make a dot or block on the screen white or black. It is a graphics command which can usually be replaced with PLOT or PRINT depending on the computer and the nature of the program.

RESTORE

The Spectrum can have a line number after the RESTORE command and this can often simplify a program. It is quite common to come across something like:

```
10 RESTORE:FOR A = 1 TO 4: READ A$: NEXT A
11 DATA "FISH","BIRDS","MAMMALS","REPTILES"
12 DATA "CATS","DOGS"
```

This is necessary to step over certain data until the right section is reached. On the Spectrum, replace everything in line 10 with:

```
10 RESTORE 12
```

RIGHT$

RIGHT$(R$,X) takes the right-hand X characters of the string R$. On the Spectrum use R$(LEN R$ − X + 1 TO). Note that nothing is necessary after the TO in the Spectrum version.

SCROLL

The ZX81's SCROLL command may be replaced with PRINT AT 21,31 ' ' (note the 2 apostrophes). This has the advantage of being in simple BASIC but has the disadvantage of having to cope with the scroll? query. The ROM call LET A = USR 3582 will scroll the screen without this effect occurring. Of course, it has the disadvantage that if ever a new ROM is issued the addresses may be different.

SET

See under RESET.

TAB(X,Y)

Apart from the fact that the program may have been intended for a computer with more characters across the screen (or less), this is the same as AT Y,X; on the Spectrum.

THEN

THEN may be omitted on some computers, e.g. IF X = 2 GOTO 10, but it must be included on the Spectrum, e.g. IF X = 2 THEN GOTO 10.

Undefined variables

If a program attempts to use a variable before it has been set up most BASICs will assume a value of zero. However, an error "2 Variable not found" is generated on the Spectrum.

VAL

Similarly, if the string to which VAL is applied is not numeric the value Ø is obtained. On the Spectrum, you may obtain different errors (e.g. error 2 as above or error C Nonsense in BASIC).

Understanding the screen display

After reading chapter 24 of the Sinclair manual you may be aware of the apparently unusual layout of the dot pattern of the screen display. The copy of the picture in memory is stored in two blocks. First is the pattern of dots stored in a 6144 byte block in addresses 16384 to 22527. Every dot on the screen has a corresponding bit (eight bits in every byte, 6144 * 8 = 49152 dots/bits) but the dots are not in the same order on the screen as they are in the memory. This will be explained later.

The second block contains the colour, brightness and flashing information in a 768 byte block in addresses 22528 to 23295. We'll take a look at both of these sections. First, the display file containing the screen dot pattern. Let's try POKEing into this to study the effect it has. Remember that each bit corresponds to a dot on the screen and when we POKE a whole byte we're changing eight bits/dots at the same time. Any bit that is set to 1 will be in INK colour on the screen and any bit that is made Ø will appear as PAPER colour. The value we'll POKE is 255 (which is BIN 1111 1111 which sets all eight bits to 1 each. In other words, it will appear as a line one character space wide and one dot high:

```
10 FOR A=16384 TO 22527
20 POKE A,255
30 NEXT A
```

You will notice several things. First of all, there does not appear to be a sensible pattern in the way the screen is filled. Second, the screen is filled in three blocks, the top eight rows (PRINT

rows 0 to 7), the middle eight rows (PRINT rows 8 to 15) and then finally the bottom eight rows (PRINT rows 16 to 23).

This means that even the lower screen is stored here. The pattern is that first of all the top line of dots in PRINT row 0 is filled in, then the top line of dots in PRINT row 1, and so on down to the top line of dots in PRINT row 7. Then the second line of dots in PRINT row 0 is filled in, then the second line of dots in PRINT row 1 is filled, and so on until the eight lines of dots in PRINT rows 0 to 7 are all filled in. From this we can conclude that to find the address of the top line of dots in any of the character rows (using the PRINT AT Y,X; co-ordinates) in row Y, column X. We could use:

```
LET ADDRESS=16384+Y*32+X
```

This would only work for values of Y from 0 to 7. You may have noticed that the middle block of 8 PRINT rows is filled in the same as the top row once that has been completed. Unfortunately, the equation only works for the top rows 0 to 7. However, this equation will tell you where to POKE for the top line of dots in PRINT rows 8 to 15:

```
LET ADDRESS=16384+2048+(Y-8)*32+
X
```

The number 2048 is how many bytes represent PRINT rows 0 to 7. Remember, there are 32 columns across the screen, 8 bytes for each character location and 8 rows in that block of the screen (32 * 8 * 8 = 2048). Similarly, we need another equation for the lower third block of the screen:

```
LET ADDRESS=16384+4096+(Y-16)*32
+X
```

These could be used like this. This is an example program which prints the number 8 down the left of the screen and then POKEs a line above each figure 8 like underlining only on top of a character:

```
  10 LET X=0
  20 FOR Y=0 TO 21
  30 PRINT AT Y,0;"8"
  40 IF Y<=7 THEN LET ADDRESS=16
384+Y*32+X
  50 IF Y>=8 AND Y<=15 THEN LET
ADDRESS=16384+2048+(Y-8)*32+X
  60 IF Y>=16 THEN LET ADDRESS=1
6384+4096+(Y-16)*32+X
  70 POKE ADDRESS,255
  80 NEXT Y
```

It is more than a bit inconvenient to have to write three lines of program (40 to 60) where one would have done. The more mathematically inclined may have seen that they could be summarised in one line like this:

```
LET ADDRESS=16384+2048*INT (Y/8)
+(Y-(INT (Y/8)*8))*32+X
```

Which after a bit of juggling comes to:

```
LET ADDRESS=16384+2048*INT (Y/8)
+32*Y-(INT (Y/8)*8*32))+X
```

Or:

```
LET ADDRESS=16384+2048*INT (Y/8)
+32*Y-256*INT (Y/8)+X
```

Or:

```
LET ADDRESS=16384+1792*INT (Y/8)
+32*Y+X
```

And which finally comes down to:

```
LET ADDRESS=16384+32*(56*INT (Y/
8)+Y)+X
```

Which looks strange, but works. So if we change lines 40, 50 and 60 to our one line wonder:

```
10 LET X=0
20 FOR Y=0 TO 21
30 PRINT AT Y,0;"8"
40 LET ADDRESS=16384+32*(56*IN
T (Y/8)+Y)+X
50 POKE ADDRESS,255
60 NEXT Y
```

That works fine. You may be wondering about what to do for lines other than the top one in any character. It turns out that as the top line of each block of eight PRINT rows are stored after each other, then the second line of dots in every PRINT row follows 256 bytes (32 characters per row * 8 rows = 256) further on, and so on.

We then go through the same process for the three blocks each of 8 rows described earlier. To cut a long story short, we end up with this equation to fill in the lines of character location Y rows down, X columns across. The values of LINE (the lines) are 0 to 7 from the top line to the bottom line. The same as you would do for the user defined graphics.

```
LET ADDRESS=16384+32*(56*INT (Y/
8)+Y)+X+LINE*256
```

Try this demo program which fills in a solid line alongside a column of eights:

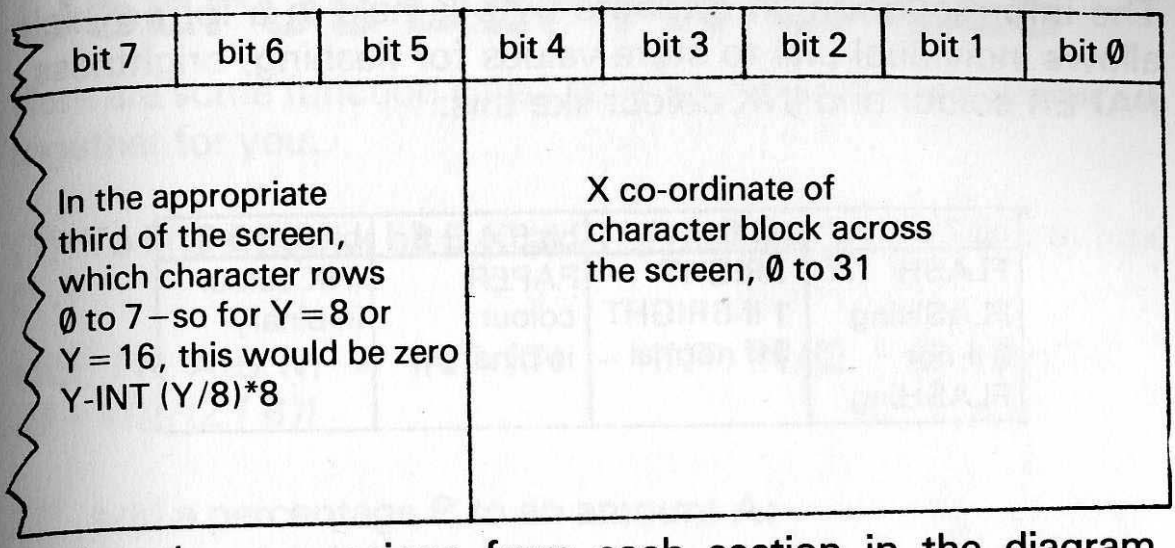
```
10 LET X=1
20 FOR Y=0 TO 21
30 PRINT AT Y,0;"8"
40 FOR L=0 TO 7
50 LET ADDRESS=16384+32*(56*IN
T (Y/8)+Y)+X+L*256
60 POKE ADDRESS,255
70 NEXT L
80 NEXT Y
```

Try this program which very slowly and inefficiently fills the screen by POKEing the addresses in the display file in the right order to fill every character location in the order the person viewing the screen might expect. Incidentally, extending the value of X to 23 means you can PLOT on lines 22 and 23 which you could not normally do. Note, however, that you cannot POKE individual dots without affecting the eight dots controlled by the entire byte, unless you are prepared to check the values already in the byte and determine accordingly what value to POKE.

```
10 FOR Y=0 TO 23
20 FOR X=0 TO 31
30 FOR L=0 TO 7
40 LET ADDRESS=16384+32*(56*IN
T (Y/8)+Y)+X+L*256
50 POKE ADDRESS,255
60 NEXT L
70 NEXT X
80 NEXT Y
```

In binary, there is an easy to understand logical method of dealing with the display file. Certain groups of bits denote certain information relating as to position in the display file. This comes in useful in machine code since it allows text and graphics access to the screen quite easily. This diagram will show how this is organised.

bit 15	bit 14	bit 13	bit 12	bit 11	bit 10	bit 9	bit 8
0	1	0	These two bits are 0 if the dot referred to is in the top third of the screen, 1 if in the middle third or 2 if in the lower third		Which line of dots in 8*8 character block-Value 0 for the top row, 7 for lowest,		
Fixed value to point to the start of the display file-BIN 0100000000000000 is 16384			2048*INT(Y/8)		256 * LINE		
16384							



Gather the expressions from each section in the diagram together and you arrive at a familiar looking expression:

16384 + 2048 * INT(Y/8) + Y – (INT(Y/8) * 8) + X + LINE*256

Secondly, the attributes file. This is much easier to understand. The colour/flashing/brightness information for any one character location is all stored in one byte. There are 32 by 24 character squares on the screen making 768 bytes in all. These are stored in a block of memory called the attributes file from addresses 22528 to 23295. The layout of these bytes is quite simple to understand compared with the display file.

The first byte (22528) corresponds to the top left character on the TV screen. The first 32 bytes correspond to the top row of 32 characters across the screen. The next 32 bytes correspond to the second row of 32 characters across the TV screen, the next 32 bytes correspond to the third row of 32 characters, and so on right down to the last character in the display on the bottom row (row 23). If you want to see how straightforward it is to understand this, try this program which will make all of the screen appear black. Note the order in which the screen is filled, including the bottom two lines. Although the bottom two lines are made black for a moment, the report code promptly overwrites this. Add a PAUSE 0 to line 30's end to see the effect and press any key (except the SHIFTS!) to finish the program:

```
10 FOR A=22528 TO 23295
20 POKE A,0
30 NEXT A
```

The information in an attribute byte is held in a form which allows individual bits to store values for flashing, brightness, PAPER colour and INK colour like this:

bit 7	bit 6	bit 5,4 & 3	bit 2,1 & Ø
FLASH 1 if FLASHing Ø if not FLASHing	BRIGHT 1 if BRIGHT Ø if normal	PAPER colour in binary	INK colour in binary

Given this, it is easily possible to work out where to POKE into any location of the attributes file if you know the PRINT AT Y,X; co-ordinates (where Y is the PRINT position down the screen and X the same across the screen).

```
POKE (22528+32*Y+X),NUMBER
```

This could also be done using PRINT AT Y,X;OVER 1;" " — but that's another story.

Try this program. It will ask you to enter the Y and X values, print a character there, then ask you to enter a value for the attributes and you will see it change colour, brightness and/or flashing depending on what you entered. From the above diagram, you may like to use BIN to enter the attribute value — e.g. if you wanted a non-flashing, bright, white PAPER and blue INK characters, you'd enter BIN Ø1111ØØ1:

```
  10 INPUT "ENTER VALUE FOR X ";
X
  20 INPUT "ENTER VALUE FOR Y ";
Y
  30 PRINT AT Y,X;"+"
  40 INPUT "ENTER VALUE FOR ATTR
IBUTES ";ATTRIBUTES
  50 POKE (22528+32*Y+X),ATTRIBU
TES
```

There's not really any point in PEEKing the attributes file as ATTR can do it for you — but if you're determined, use this:

```
LET P=PEEK (22528+32*Y+X)
```

USEFUL DEF FN CALLS

Here are some function calls I have found useful and gathered together for you.

(1) To find whether bit B (Ø to 7) of number N (Ø to 255) is 1 or Ø:

```
DEF FN A(B,N) = INT ((N - INT (N/(2 ↑ (B+1)))*(2 ↑ (B+1)))/(2 ↑ B))
```

(2) Add a percentage P to an amount A:

```
DEF FN S(A,P) = A * P/1ØØ + A
```

(3) Given a total amount T which includes a percentage P added to an original amount, this function will tell you what the original amount was. Useful for VAT calculations, etc:

```
DEF FN A(P,T) = (T * 1ØØ)/(1ØØ + P)
```

(4) Odd or even. FN O() will be 1 if the number N is odd, or zero if it is even:

```
DEF FN O(N) = N - INT (N/2)
```

(5) To find the PAPER colour at screen location Y,X (i.e. resolve the attributes to components):

```
DEF FN P(Y,X) = INT ((ATTR (Y,X) - INT (ATTR (Y,X)/64)*64)/8)
```

(6) To find the INK colour at screen location Y,X:

```
DEF FN I(Y,X) = INT (ATTR (Y,X) - INT (ATTR (Y,X)/8) * 8)
```

(7) To find FLASH state (Ø or 1) at screen location Y,X:

```
DEF FN F(Y,X) = INT (ATTR (Y,X)/128)
```

(8) To find the state of BRIGHT at screen location Y,X:

```
DEF FN B(Y,X) = INT ((ATTR (Y,X) - INT (ATTR (Y,X)/128) * 128)/64)
```

(9) 2 byte PEEK starting at address A:

```
DEF FN P(A) = PEEK A + 256 * PEEK (A + 1)
```

You could use this in your program while typing it in. At any time, PRINT FN(23641) - 16384 will give you a vague idea of how much memory has been used up (including everything up to the end of the variables). This saves typing in a lengthy PEEK expression all the time.

(10) Memory left in bytes:

```
DEF FN M() = PEEK 23730 + 256 * PEEK 23731 - PEEK
23653 - 256 * PEEK 23654
```

(11) Round off the value of X to the nearest whole number:

DEF FN W(X) = INT (X + Ø·5). This suffers from the way the Spectrum holds the number Ø·5 in such a way that giving X a value of Ø·5 will yield Ø. This can be corrected by increasing the value added to X in the function to, say, Ø·5ØØØØØØ1.

(12) Random numbers between 1 and X:

```
DEF FN R(X) = INT (RND * X) + 1
```

(13) Centering a string around the middle of the screen. This returns the value for TAB to print the string so that it is in the middle of the screen:

```
DEF FN M(M$) = (16 - LEN M$ / 2) AND LEN M$ < 33
```

This only works for strings up to 32 characters long — if longer than 32 characters, the routine will give up and just print from the left of the screen. This is how to use the function — put it in a PRINT statement as the argument of TAB remembering to place the string to be printed in the brackets, e.g. PRINT TAB FN M(A$);A$ or PRINT TAB FN M("DOG");"DOG".

(14) To find the log base 1Ø of a number N using the Spectrum's natural logs:

```
DEF FN L(N) = LN N/LN 1Ø
```

The formula can be used to find logs to other bases, e.g. base B:

```
DEF FN L(N,B) = LN N/LN B
```

(15) Rounding off an amount A to 2 demical places:

```
DEF FN R(A) = INT (A * 100 + 0·5)/100
```

Rounding off an amount A to D decimal places:

```
DEF FN R(A,D) = INT (A * (10 ↑ D) + 0·5)/(10 ↑ D).
```

INPUT-OUTPUT CHANNELS

It is possible to use certain commands to provide OUTPUT to different destinations or to accept INPUT from certain sources, e.g. try INPUT #2 ; "This is #2"; #1;X.

This introduces us to the system of channels used for the INPUTting and OUTPUTting of information on the Spectrum. There are 19 possible channels. Channels −3 to −1 are used by the operating system but cannot practically be used by the programmer. On the bare Spectrum (no expansion module, microdrives, etc.) channels Ø to +3 are implemented as follows. Note how only channel 1 allows INPUT normally:

Channel	INPUT	OUTPUT
Ø	none	to workspace/lower screen
1	keyboard	editing area/lower screen
2	none	upper screen
3	none	printer

Channels 4 to 15 are not used at the time of writing (unless specially opened by the user) although they will probably be used for some of the expansion options such as RS232, microdrives and networking. We can make use of certain channels for printing, e.g. try:

```
10 FOR A=Ø TO 3
20 PRINT #A;"THIS IS #";A
30 NEXT A: PAUSE Ø
```

Note how PRINT output can go to both parts of the screen or even to the printer. PRINT is not the only command that can be used in this way. Try:

```
10 INPUT #2;"Enter a number ";
#1;X
```

The prompt message string for INPUT appears in the upper screen (note that it is not erased afterwards) whereas anything you type appears in the normal place. This is because the keyboard INPUT can only come from channel 1 as set up by the ROM. OUTPUT can go to any channel specified. LLIST can also be replaced by PRINT #3 and LPRINT replaced by PRINT #3. Try LPRINT #2 — who needs PRINT anyway? The most obvious application for this is to use PRINT #1 to enable us to print on those lower two screen lines. To see what happens when you try to get INPUT from channels Ø,2 or 3, try INPUT #3;X.

That example tried to get INPUT from the printer which you can't do, obviously. An error J Invalid I/O Device occurs. You can open a channel with the OPEN # statement. To do this you must specify what device is attached to which channel. After the # comes the channel number, then a comma, then in quotes a letter to denote the "device"; at the moment R, K, S or P for workspace/lower screen, editing area/lower screen & keyboard, upper screen and printer respectively, e.g. OPEN 5,"P". The same can be done to close a channel using CLOSE #5, for example. Beware of trying to CLOSE a channel before it has been opened as spurious results occur.

CONTROLLING THOSE NUMBERS

Printing the numbers to the screen can often be an art in itself if any control is needed over those numbers. Serious programs especially require that numbers be tabulated in a certain manner. This could involve making sure that only a certain number of decimal places are printed, that all the numbers are printed beneath one another with decimal points aligned, or that all the numbers contain the same amount of digits.

All of these usually demand that the number be turned into a string then examined character by character to find the decimal point, the integer part studied to decide how many zeros to add, or the length checked, and so on. I'll look at routines to provide the various type of formatting often used.

(1) Round off a number to a set number of decimal places

Numerically it is quite simple to round off an amount (AMOUNT) to 2 decimal places. All that is needed is a formula like:

LET TWODEC = INT (AMOUNT*100+0·5)/100

```
LET TWODEC=INT (AMOUNT*100+0.5)/
100
```

This is very useful for money values since you can round off to the nearest penny. This short program will show what the routine can do and some of its shortcomings. Line 10 generates an amount AMOUNT at random which line 20 reduces to TWODEC, a copy of AMOUNT rounded off to two decimal places. Both are printed side by side in line 30 for comparison.

```
  10 LET AMOUNT=RND*100
  20 LET TWODEC=INT (AMOUNT*100+
0.5)/100
  30 PRINT AMOUNT,TWODEC
  40 GO TO 10
```

```
34.892273       34.89
16.993713       16.99
74.623108       74.62
96.762085       96.76
57.159424       57.16
87.005615       87.01
25.434875       25.43
7.699585        7.7
77.574158       77.57
18.086243       18.09
56.561279       56.56
42.144775       42.14
60.923767       60.92
69.326782       69.33
99.543762       99.54
65.782166       65.78
33.700562       33.7
27.616882       27.62
71.348572       71.35
51.174927       51.17
38.174438       38.17
63.153076       63.15
```

The routine works quite well, if somewhat untidily. Numbers less than 0·1 are printed without zeros before the decimal point and numbers greater than or equal to 0·1 and less than 1 are printed with a zero before the decimal point. There can also be a variable amount of digits after the decimal point, e.g. the number 2 (no digits after the decimal point), 2·2 (1 digit after the decimal point) and 2·24 (which has two digits after the decimal point).

If you wish to round off an amount AMOUNT to PLACES decimal places, use this routine. ROUNDED is AMOUNT once it has been rounded off:

LET ROUNDED = INT (AMOUNT*(10 ↑ PLACES)+0·5)/(10↑PLACES)

```
LET ROUNDED=INT (AMOUNT*(10↑PLAC
ES)+0.5)/(10↑PLACES)
```

The use of exponentiation ↑ makes the routine slow compared to the version before. If this latest routine is to be used more than once, try storing the exponentiation in another variable to speed things up slightly:

LET MULT=10 ↑ D: LET ROUNDED=INT (AMOUNT*MULT+0·5)/MULT

```
LET MULT=10↑D: LET ROUNDED=INT (
AMOUNT*MULT+0.5)/MULT
```

(2) Rounding off to a fixed number of decimal places, with zeros and/or decimal point added as necessary

Here, AMOUNT is the amount to be printed to 2 decimal places. This ends up in string form as A$, a string with any additional digits added as needed.

```
   10 LET AMOUNT=RND*100
   20 LET A$=STR$ (INT (AMOUNT*10
0+0.5)/100)
   30 IF A$(1)="." THEN LET A$="0
"+A$
   40 LET C=LEN A$-LEN STR$ INT V
AL A$
   50 LET A$=A$+".00"(C+1 TO )
   60 PRINT "£";AMOUNT,"£";A$
   70 GO TO 10
```

```
£8.581543       £8.58
£43.719462      £43.72
£79.025269      £79.03
£26.91803       £26.92
£18.934631      £18.93
£20.188904      £20.19
£14.257813      £14.26
£69.433594      £69.43
£7.5531006      £7.55
£66.58783       £66.59
£94.125366      £94.13
£59.408569      £59.41
£55.688477      £55.69
£76.686096      £76.69
£51.483154      £51.48
£61.291504      £61.29
£96.907043      £96.91
£68.031311      £68.03
£2.3834229      £2.38
£78.868103      £78.87
£15.130615      £15.13
£34.892273      £34.89
```

The output from this program is what you might expect to see on a price tag, in pounds. Line 10 sets up the initial amount and line 20 converts this to a string equivalent rounded off to 2 decimal places. Line 30 adds a zero at the beginning, if needed. Line 40 finds the length of the whole number part subtracted from the entire length, i.e. digits after decimal point if any. This is used to determine how many zeros/decimal points are to be added in line 50. Line 60 prints both versions side by side to compare.

(3) Lining up decimal points

Where you have charts of numbers, a faster visual understanding is obtained if the numbers are aligned in such a way that the decimal points all line up under one another.

```
10 LET AMOUNT=RND*1000
20 PRINT AMOUNT
30 GO TO 10
```

```
1.1291504
85.81543
437.19482
790.25269
269.1803
169.34631
201.88904
142.57813
694.33594
75.531006
665.8783
941.25366
594.08569
556.88477
766.86096
514.83154
612.91504
969.07043
680.31311
23.834229
788.68103
151.30615
```

To line up the decimal points, all that is needed is to change line 20 slightly:

```
   10 LET AMOUNT=RND*1000
   20 PRINT TAB 15-LEN STR$ INT A
MOUNT;AMOUNT
   30 GO TO 10
```

```
   169.93713
   746.23108
   967.62085
   571.59424
   870.05615
   254.34875
    76.99585
   775.74158
   180.86243
   565.61279
   421.44775
   609.23767
   693.26782
   995.43762
   657.82166
   337.00562
   276.16882
```

```
713.48572
511.74927
381.74438
631.53076
365.21912
```

See how much tidier this is? If you wish to align decimal points of numbers formatted to 2 decimal places held in a string, you cannot always use the above method because the string may be affected by converting with VAL. However, you know how many decimal places there will be, so working from the right:

```
20 PRINT TAB 15-LEN A$;A$
```

Note that a rather nasty effect occurs when numbers get so large that STR$ starts using scientific notation. Of course, unless you were dealing with amounts greater than a hundred millions, this would not bother you.

(4) Overwriting columns of numbers

When tabulating numbers in columns, it is common to over-print numbers on top of one another to update values. This runs the risk of one value having a different number of digits to another, leaving some of them on the screen. As a simple example, try:

```
10 FOR A=110 TO 0 STEP -1
20 PRINT AT 0,0;A
30 NEXT A
```

The numbers printed at the top of the screen start off as three digit numbers, but once the numbers become 2 digit numbers a zero remains on the right of the column(s). This could be avoided by printing a few spaces after the number. Of course, if you've got graphics/text/another column of numbers alongside, this will be erased as well.

Use this routine to print only enough spaces to reach just as far as the longest (in terms of digits) number. Line 20 should have in the print quotes one less space than the maximum number of digits to be printed. The number after TO should be that maximum amount of digits.

```
  10 FOR A=110 TO 0 STEP -1
  20 PRINT AT 0,0;A;"   "( TO 3-L
EN STR$ A)
  30 NEXT A
```

ROM ROUTINES

There are a number of useful routines in the 16K ROM that can used in user-written machine code programs. This list details some of them, but is by no means exhaustive since this would be the subject of an entire book in itself. All addresses referred to are in decimal unless otherwise stated.

16 PRINT routine. The contents of the A register is sent to the current OUTPUT channel. If the current channel is anything other than the one that you want to PRINT to, use the routine at 5633 with the A register holding the number of the channel to be opened (normally upper screen, channel 2). You can use the PRINT routine to OUTPUT control characters as well as the normal characters to be printed.

949 BEEP. This will sound a note on the loudspeaker and the tape sockets like the BASIC BEEP command. On entry to this routine, register pair HL should contain the frequency (lower values give higher frequency) and register pair DE should contain the duration of the note (lower values give shorter durations). Note that the frequency affects the duration of the note so that if you doubled the frequency and had the same duration value, the note would not necessarily be the same length. Calling this routine repeatedly with a short duration BEEP of different frequencies could simulate sliding tones or even envelope control, something that is not possible in BASIC.

3435 CLS. Use of this routine is the same as the BASIC command CLS.

3582 Scroll the screen. USR 3582 or CALL 3582 will scroll the screen up by one character line without affecting the PRINT position or affecting the SCR_CT system variable. Very useful for games where a continuous scrolling action is required without having to use something like:

POKE 23692,255: PRINT AT 21,31''

which forces a screen scroll and suppresses the scroll? query.

3742 Address of the start of the line of display. If the A register is loaded with the PRINT row number (Ø to 23 down the

screen) the value of HL on return from this routine will be the address of the top line of dots in the first character on that line.

3756 The same as the BASIC command COPY. The interesting aspect is that after disabling the interrupts, it starts by specifying how many lines in the upper screen will be copied to the printer. So, you can replace this part with your own routine to specify how many lines of the upper screen are to be copied into the printer. This number should be in the B register. The routine might look like this:

```
DI                  : disable interrupts
LD B,number         : how many lines to be copied
CALL 3759           : COPY it to printer
```

3789 LPRINT. Prints whatever is in the printer buffer to the ZX Printer and resets everything concerned with the printer buffer.

6683 Prints contents of register pair BC as a decimal number. Ø to 9999 are the values allowed because it is normally used for printing line numbers.

7997 PAUSE Register BC is to hold the time delay in 50ths of a second in the UK. Like the BASIC PAUSE command, this delay is ended either when the time delay ends or a key is pressed. Like PAUSE Ø, if BC contains zero on entry, the C register contains the X co-ordinate and the B register contains the Y co-ordinate. On returning, the number in the A register denotes which bit of the address in HL corresponds to the pixel X,Y on the screen.

8252 PRINT bytes. This routine will print to the current output channel a string starting at the address held in register pair DE of length stored in reg. pair BC.

8855 border colour. To change border colour put the colour number in the A register the call 8855.

8874 This routine will tell you which bit of which address in the display file corresponds to which screen pixel. On entry the C register contains the X coordinate and the B register contains the Y coordinate. On returning, the number in the A register denotes which bit of the address in HL corresponds to the pixel X,Y on the screen.

8927 PLOT routine. Register B should contain the Y co-ordinate, register C the X co-ordinate on entry. Corresponds to PLOT X,Y.

9402 DRAW straight lines. Four registers need to be set up as follows, before calling the routine. Load the B register with the Y offset (absolute value) and load the C register with the X offset (absolute value). Then, registers D and E have to contain 1 for positive or 255 for negative offsets (the SGN of the offsets). Register D will denote the sign of the X offset and register E the sign of the Y offset.

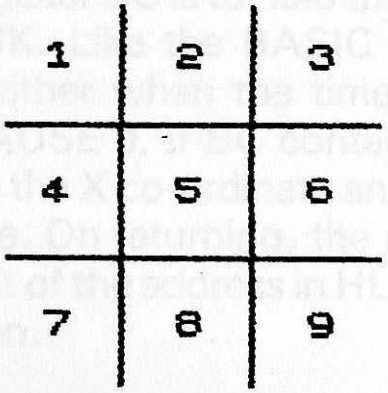
NOUGHTS & CROSSES

This program plays the game of noughts and crosses in which the object is for you or the computer to take alternate turns to try to get three Os or three Xs respectively in a row up, across, or diagonally. Unlike many programs to play this game, the computer can be beaten — but not all the time. The program plays defensively, so most games tend to end in a draw.

When RUN, the program asks if you want to have first go. Answer y if you want first go, or n if you don't. You should ensure that CAPS LOCK is off, as the program only recognises lower case letters. If you add the line:

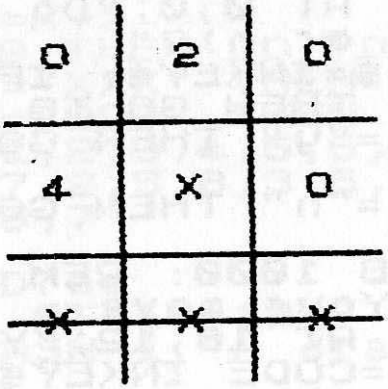
```
15 POKE 23658,0
```

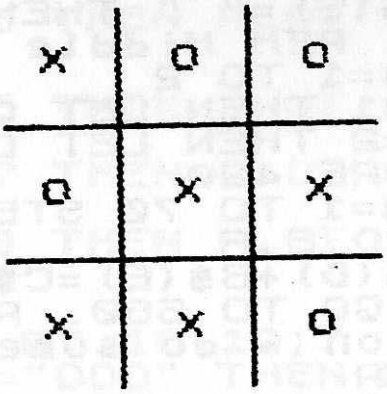
this would allow the computer to switch off the CAPS LOCK by itself. The program uses INKEY$ to detect keyboard responses, so you don't have to press ENTER at any time except when running the program. The board is numbered as follows:



To make your move, press the key with the same number as the square on which you wish to place an O; the computer is always X. Once a winning line has been made, the computer draws a line through that winning line and flashes on the screen who has won. Example displays and listing follow:

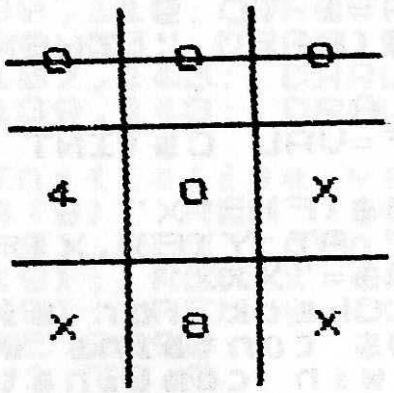
I win





A draw

You win

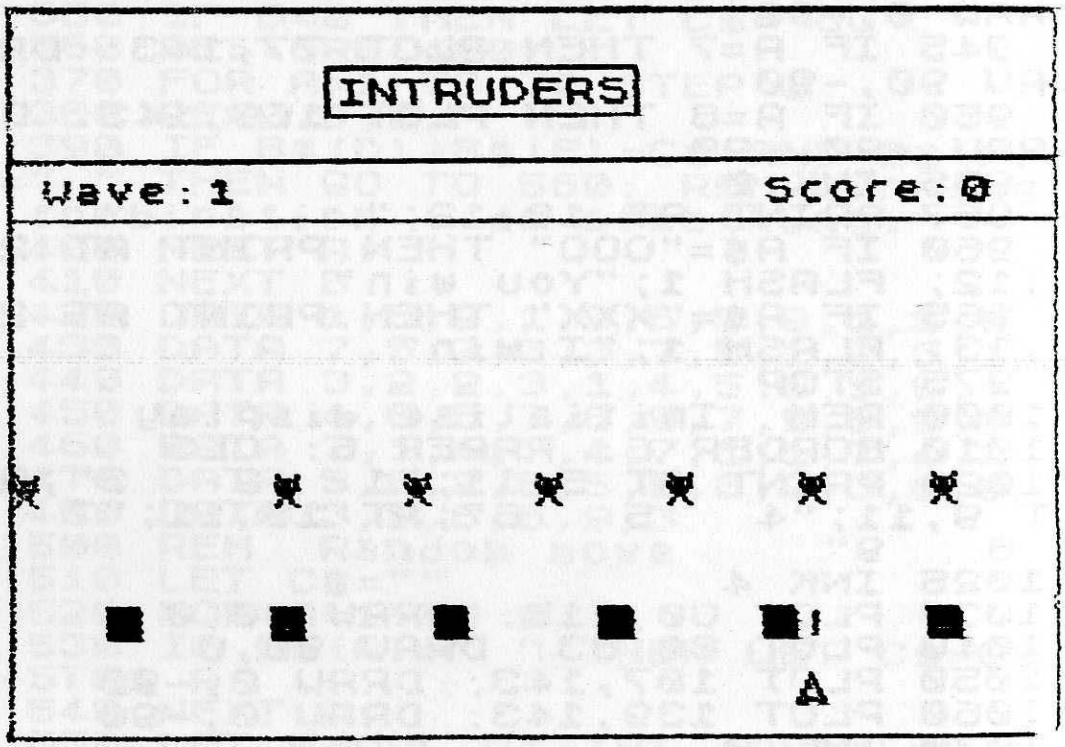


```
    0>REM . Noughts and crosses
    0>REM    by   Dilwyn Jones.
   10 PRINT AT 0,0;"Do you want f
irst go (y or n)?"
   20 LET A$=INKEY$: IF A$<>"y" A
ND a$<>"n" THEN GO TO 20
   30 IF a$="y" THEN GO SUB 1000:
 GO TO 100
   40 IF a$="n" THEN GO SUB 1000:
 GO TO 300
   90 GO SUB 1000: REM Initialise
  100 REM  Your move
  110 PRINT AT 18,12;"Your go"
  120 LET A=CODE INKEY$-48: IF A<
1 OR A>9 THEN GO TO 120
  130 IF B$(A)<>" " THEN GO TO 12
0
  140 LET B$(A)="O"
  150 PRINT AT Y(A),X(A);B$(A)
  160 LET A$="OOO"
  170 GO TO 600: REM  Win?
  300 REM  Spectrum move
  310 PRINT AT 18,12;" My go "
  315 LET A$="XXX"
  320 IF B$(5)=" " THEN LET F=5:
GO TO 560: REM Middle of board
  330 FOR B=1 TO 2
  340 IF B=1 THEN LET C$="XX"
  350 IF B=2 THEN LET C$="OO"
  360 RESTORE 420
  370 FOR A=1 TO 70 STEP 3
  380 READ D,E,F
  390 IF B$(D)+B$(E)=C$ AND B$(F)
=" " THEN GO TO 560: REM Winning
 combination - do something.
  400 NEXT A
  410 NEXT B
  420 DATA 1,5,9,1,9,5,5,9,1,3,5
  430 DATA 7,5,7,3,3,7,5,1,2,3,1
  440 DATA 2,2,2,3,1,4,5,5,5,5,4
  450 DATA 4,6,5,7,8,9,7,9,8,9,8
  460 DATA 7,1,4,7,4,7,1,1,7,4,2
  470 DATA 5,8,2,8,5,5,8,2,3,6,9
  480 DATA 3,9,6,6,9,3
  500 REM  Random move
  510 LET C$=""
  520 FOR A=1 TO 9
  530 IF B$(A)=" " THEN LET C$=C$
+STR$ A
  540 NEXT A
  550 LET F=VAL C$(INT (RND*LEN C
$)+1)
  560 LET B$(F)="X"
  570 PRINT AT Y(F),X(F);B$(F)
  580 LET A$="XXX"
  600 REM  Check for Win
  601 REM A$ contains who wins if
  602 REM  win combination found
  610 RESTORE 700
  620 FOR A=1 TO 8
  630 READ D,E,F
```

```
 640 IF B$(D)+B$(E)+B$(F)=A$ THE
N GO TO 900
 650 NEXT A
 660 LET X=X+1
 670 IF X=9 THEN GO TO 800
 680 IF A$="XXX" THEN GO TO 100
 690 IF A$="000" THEN GO TO 300
 700 DATA 1,2,3,4,5,6,7,8,9,1,4
 710 DATA 7,2,5,8,3,6,9,1,5,9,3
 720 DATA 5,7
 730 STOP
 800 REM  Draw
 810 PRINT AT 18,12;"          ";AT
 21,11; FLASH 1;"A draw"
 820 STOP
 900 REM  Win
 905 INK 4
 915 IF A=1 THEN PLOT 80,131: DR
AW 90,0
 920 IF A=2 THEN PLOT 80,99: DRA
W 90,0
 925 IF A=3 THEN PLOT 80,67: DRA
W 90,0
 930 IF A=4 THEN PLOT 91,143: DR
AW 0,-90
 935 IF A=5 THEN PLOT 123,143: D
RAW 0,-90
 940 IF A=6 THEN PLOT 155,143: D
RAW 0,-90
 945 IF A=7 THEN PLOT 77,143: DR
AW 90,-90
 950 IF A=8 THEN PLOT 169,143: D
RAW -90,-90
 955 INK 0
 957 PRINT AT 18,12;"          "
 960 IF A$="000" THEN PRINT AT 2
,12; FLASH 1;"You win"
 965 IF A$="XXX" THEN PRINT AT 2
,13; FLASH 1;"I win"
 975 STOP
1000 REM  Initialise display
1010 BORDER 6: PAPER 6: CLS
1020 PRINT AT 5,11;"1   2   3";A
T 9,11;"4   5   6";AT 13,11;"7
  8   9"
1025 INK 4
1030 PLOT 80,115: DRAW 90,0
1040 PLOT 80,83: DRAW 90,0
1050 PLOT 107,143: DRAW 0,-90
1060 PLOT 139,143: DRAW 0,-90
1070 INK 0
2000 REM  Initialise variables
2010 DIM B$(9): REM  board
2020 LET X=0: REM count moves
2030 DIM X(9): REM x co-ords
2040 DIM Y(9): REM y co-ords
2050 RESTORE 2090
2060 FOR A=1 TO 9
2070 READ X(A),Y(A)
2080 NEXT A
2090 DATA 11,5,15,5,19,5,11,9,15
2100 DATA 9,19,9,11,13,15,13,19
2110 DATA 19
2120 RETURN
```

INTRUDERS

A swarm of aliens is descending on you — you must prevent them landing at all costs or they will destroy Earth. You must stand up to them alone armed only with your laser gun and protected only by a few shields. The aliens come at you in battle formation firing their deadly missiles at you with uncanny accuracy. Avoid them at all costs or hide behind your shields — but beware, they vaporise after being hit once. Align yourself with the alien and vaporise it by firing your laser cannon; you will see the alien turn white hot as the laser beam strikes it. The laser beam will do the same thing to your shields so don't vaporise them! You can't win — they'll get you eventually, but try to get a higher score than anyone else. Use the 5 (cursor left) key to move you left, the 8 (cursor right) key to move you right and the 7 (cursor up) key to fire your laser cannon up at the alien intruders. If any aliens disappear off the edge of the screen they'll promptly reappear on the other side, so don't get complacent.



```
  0>REM  Intruders
         By Dilwyn Jones
 10 GO SUB 9000
 20 GO SUB 9500
 30 FOR D=1 TO 7
 34 LET C$="    ■    ■    ■    ■
   ■    ■    "
 35 PRINT INK 2;AT 16,0;C$
 40 LET A$="¥    ¥    ¥    ¥    ¥
¥    ¥    "
 70 PRINT AT 5,6;D
 80 FOR B=D+9 TO 19 STEP 2
 85 IF B>=16 THEN LET C$=B$: PR
INT AT 16,0;C$
 90 FOR E=1 TO 3
100 LET A$=A$(32)+A$( TO 31)
110 FOR A=0 TO DIFF
111 IF A$(X+1)="¥" AND RND<.05
AND NOT BOMBY THEN LET BOMBY=B:
LET BOMBX=X
112 IF BOMBY>18 AND BOMBX=X THE
N GO TO 3500
113>IF BOMBY THEN GO SUB 3000
120 LET X=X+(INKEY$="8" AND X<3
1)-(INKEY$="5" AND X>0)
130 PRINT AT B,0; INK 4;A$; INK
3;AT 20,C;" " AND C<>X;AT 20,X;
"A"
140 IF A$=B$ THEN GO TO 240
150 LET C=X
160 IF INKEY$="7" THEN GO SUB 4
000
170 NEXT A
180 NEXT E
190 PRINT AT B,0;B$
200 NEXT B
210 PRINT PAPER 2; FLASH 1;AT 7
,5;"Oops....they landed!!!"
220 IF INKEY$<>"" THEN GO TO 22
0
230 STOP
250 NEXT D
260 LET DIFF=DIFF-1: IF DIFF<5
THEN LET DIFF=20
300 GO TO 30
1000 REM  Make a noise!
1020 BEEP .01,30: BEEP .01,40
1100 RETURN
2000 REM  More noise
2010 FOR A=0 TO 21 STEP 2
2020 BEEP 0.01,(40-(A/2))
2030 NEXT A
2040 RETURN
3000 REM  Bomb
3010 LET BOMBY=BOMBY+1
3015 IF BOMBY=21 THEN PRINT AT B
OMBY-1,BOMBX;" ": LET BOMBY=0: R
ETURN
3020 IF BOMBY=16 AND C$(BOMBX+1)
="■" THEN GO SUB 3100
3030 IF BOMBY-1>B THEN PRINT AT
BOMBY-1,BOMBX;" "
3040 IF BOMBY THEN PRINT AT BOMB
Y,BOMBX;"¦"
```

```
3050 RETURN
3100 REM  Knock out shield
3110 LET C$(BOMBX+1)=" "
3115 IF BOMBY-1<>6 THEN PRINT AT
 BOMBY-1,BOMBX;" "
3120 LET BOMBY=0
3130 PRINT AT 18,BOMBX;" "
3135 GO SUB 1000
3140 RETURN
3500 REM  Oops...bang
3510 PRINT AT BOMBY,BOMBX;" ";AT
 20,X; FLASH 1; INK 7;"A": GO SU
B 2000: PRINT AT 21,0;"Oh dear..
."
3520 STOP

4000 REM  Destroy shield
4010 IF C$(X+1)=" " THEN GO TO 5
000
4020 INK 9
4030 PLOT X*8+3,16: DRAW 0,8
4040 LET C$(X+1)=" "
4050 PLOT OVER 1;X*8+3,16: DRAW
OVER 1;0,8
4055 GO SUB 1000
4060 PRINT AT 18,X;" "
4070 INK 6
4080 RETURN
5000 REM  Fire laser beam
5005 IF a$(X+1)="▚" THEN LET SCO
RE=SCORE+1: IF SCORE>999 THEN LE
T SCORE=0
5010 INK 9
5020 PLOT X*8+3,16
5030 DRAW 0,(19-B)*8
5040 OVER 1
5050 PLOT X*8+3,16
5060 DRAW 0,(19-B)*8
5070 INK 6
5080 OVER 0
5085 GO SUB 1000
5090 LET A$(X+1)=" "
5095 PRINT AT 5,29; INK 6;SCORE
5100 RETURN
9000 REM  Initialise UDGs ▚A
9010 RESTORE 9080
9020 FOR C=0 TO 7
9030 READ A,B
9040 POKE USR "A"+C,A
9050 POKE USR "B"+C,B
9060 NEXT C
9070 RETURN
9080 DATA 66,16,60,16,90,40,126,
40,60,68,24,68,36,186,66,198
9500 REM  Initialise
9510 PAPER 0
9520 BORDER 4
9530 CLS
9540 INK 3
9550 PLOT 0,139
9560 DRAW 255,0
9570 PLOT 0,124
9580 DRAW 255,0
9590 INK 6
```

```
9600 PRINT FLASH 1;AT 2,10;"INTR
UDERS"
9610 PLOT 78,161
9620 DRAW 75,0
9630 DRAW 0,-11
9640 DRAW -75,0
9650 DRAW 0,11
9660 PRINT AT 5,1;"Wave:0";TAB 2
3;"Score:0"
9670 DIM A$(32)
9680 DIM B$(32)
9690 DIM C$(32)
9695 LET BOMBY=0: LET BOMBX=0
9696 LET X=INT (RND*32): LET C=X
9700 LET SCORE=0
9705 LET DIFF=20
9710 RETURN
```

The program requires some 12K including screen, variables, program and user defined graphics to RUN, and so is OK for the 16K Spectrum. It is an arcade-style game written in BASIC and makes use of several colours, high resolution graphics, user defined graphics and sound for exciting fast action. It has a score facility to provide an element of competition. Extensive use is made of strings for fast handling of data and the PLOT, DRAW and OVER commands are used for line drawing and erasing. The program proper is contained in lines 30 to 300 and the rest of the program is mainly subroutines to perform various functions. All are marked with REMs to identify them. All that needs adding is to note the graphics characters in each line:

34:	graphics SHIFT 8
40:	graphics A
111:	graphics A
130:	graphics B
3020:	graphics SHIFT 8
3510:	graphics B
5005:	graphics A
9000:	graphics A followed by graphics B

If you intend to change the program at all, note that it has been written to be fast, so don't do anything to slow it down. The sound commands in the subroutine 1000 should be kept very short — about 0·05 seconds maximum. Do what you like to the other sound subroutine — it only ever occurs at the end of the program when you've been defeated, so it's been made long to humiliate!

SUPER SOUNDS

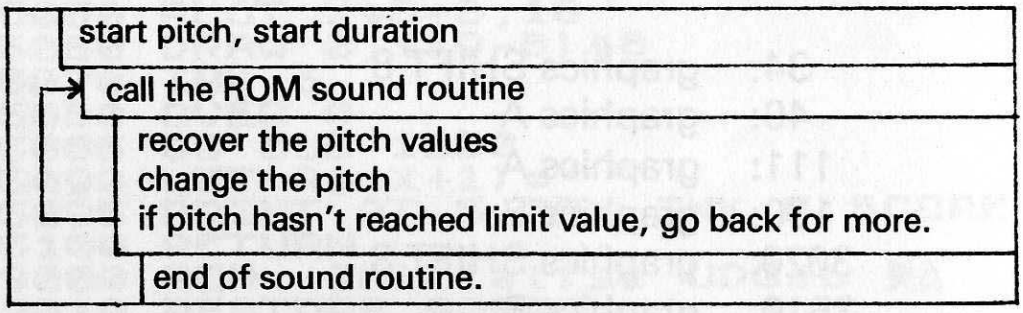
In BASIC the Spectrum's sound facilities are limited to creating a single note of fixed duration and pitch like:

BEEP duration, pitch

We can extend this slightly by playing several short notes quickly after one another, for example:

```
10 FOR A=0 TO 21
20 BEEP 0.01,(40-(A/2))
30 NEXT A
```

However, we are forced to the conclusion that to produce any more complex sounds, BASIC is not capable of doing so. Even when we turn to machine code, we are still limited to control over pitch and duration of a sound. However, what machine code can do is speed up the process a great deal so that we could sound a sequence of notes, one after the other, so quickly they could pass for one longer note. And if we arranged that the notes are slightly different in pitch, we could make some notes not normally possible (think of the synthesised drum "peeow" sound to be found on pop records). The way to do this is to use the ROM routine that normally handles the BEEP command by repeatedly calling with very short notes of changing frequencies. Here is a diagram to illustrate:



Here is a machine code program which will allow us to do this. If you don't understand machine code or assembler language, full instructions will be given so that you can run the routines on your 16K or 48K Spectrum. These sounds can be used in any program you care to add these routines to.

HEXA-DECIMAL	DECIMAL	ASSEMBLER	REMARKS
06 0A	6,10	LD B,10	; counter for repeats
C5	197	COUNT;PUSH BC	; save counter on stack
21 00 00	33,0,0	LD HL,0	; starting pitch
11 64 00	17,100,0	LOOP ;LD DE,100	; duration of one sound
E5	229	PUSH HL	; save the pitch value

```
CD B5 03    205,181,3    CALL 949                  ; call ROM BEEP routine
01 14 00    1,20,0       LD BC,20                  ; pitch change step value
11 64 00    17,100,0     LD DE,100                 ; pitch limit value
E1          225          POP HL                    ; recover pitch from stack
C6 00       198,0        ADD A,0                   ; reset carry flag
ED 4A       237,74       ADC HL,BC                 ; new pitch value
E5          229          PUSH HL                   ; save new pitch on stack
C6 00       198,0        ADD A,0                   ; reset carry flag
ED 52       237,82       SBC HL,DE                 ; test for pitch limit
E1          225          POP HL                    ; recover pitch value
38 E6       56,230       JR C,LOOP (JR C, -26)     ; pitch limit reached?
C1          193          POP BC                    ; recover counter of repeats
10 DF       16,223       DJ NZ,COUNT (DJ NZ, -33)  ; more repeats?
C9          201          RET                       ; return to BASIC
```

Relative jumps have been shown in assembler as both jumps to labels for clarity and relative (minus) jumps for practical purposes. The values given for numbers loaded into registers are examples. The remarks alongside the assembler will show what goes where.

Briefly, the above 36 byte routine takes a note of pitch HL and duration DE and repeatedly sounds it with the pitch descending until a limit value is reached, then repeats this the number of times specified. Note that when using the ROM BEEP routine in this way, the duration value is dependent on the pitch value (think of it as number of cycles rather than duration — a higher pitch has a shorter period).

Both pitch and duration can start at 1 for high pitch and short duration respectively but going for too large a value for either will result in very long low pitched useless tones. What I'll do is give you a decimal machine code loader which will read the values of the bytes of machine code from DATA statements to be POKEd into memory above RAMTOP. You need to reserve 36 bytes for one of these routines. For 16K users, I suggest that you CLEAR 32563 so that the routine will start at 32564. For 48K users, I suggest that you CLEAR 65331 in the first line of the load program so that the routine will start at 65332. Two versions of the loader follow, one for 16K and another for 48K:

```
  1 REM  48K SOUNDS LOADER
 10 CLEAR 65331
 20 LET ADDRESS=65332: LET END=
1000
 30 READ BYTE: IF BYTE=END THEN
STOP
 40 IF BYTE>255 THEN STOP
 50 POKE ADDRESS,BYTE
```

```
60 LET ADDRESS=ADDRESS+1
70 GO TO 30

  1 REM  16K SOUNDS LOADER
 10 CLEAR 32563
 20 LET ADDRESS=32564: LET END=
1000
 30 READ BYTE: IF BYTE=END THEN
 STOP
 40 IF BYTE>255 THEN STOP
 50 POKE ADDRESS,BYTE
 60 LET ADDRESS=ADDRESS+1
 70 GO TO 30
```

Line 10 sets up the new RAMTOP above which will go the machine code. You will need to adjust this if you make the machine code longer (e.g. combine more than one, in which case, for the examples given you will need 36 bytes each). To determine this, find out the number in the system variable RAMTOP (normally 32599 on a 16K Spectrum, 65367 on a 48K Spectrum) then subtract the number of bytes of machine code (e.g. for one of these routines on a 16K Spectrum 32599 − 36 = 32563) whereupon the machine code will start immediately after this new RAMTOP value. This brings us to line 20. ADDRESS is where the routine will start, just after RAMTOP as described. The variable END is used as a marker for the end of the DATA list of numbers. All it means is that numbers to be POKEd must have a value 0 to 255; so reading a value greater than 255 can be taken to be a signal that the end of the list has been reached. It could be done simply with any number, but assigning a variable and using it as a word gives an immediate visual indication (note that not every computer would allow you to read a variable name from the DATA list like this). This is done in lines 30 and 40. Line 50 puts the bytes of machine code in place in memory, then line 60 increments the address for the next byte.

Here are some examples of the sounds that can be generated. These are the DATA statements to be used with the loader program. They all contain 36 bytes of the same machine code, but with different pitch, repeats, etc. Type in the DATA statement of your choice and run the loader program. The program should stop with report 9 STOP statement, 30:3. If it doesn't and you've not made any changes to the program, there's a mistake somewhere. SAVE the program on tape in case you

lose it when trying to run the machine code. When you are ready to hear the sounds, once the machine code has been set up (and ultimately this is the only way to be sure it's all OK once you've saved it all on tape in case), you can use these commands to run the machine code:

LET A = USR 32564 (for 16K Spectrum)

LET A = USR 65332 (for 48K Spectrum)

Of course, you could use something like RANDOMIZE USR 65332, but this may affect randomness of random numbers generated if, as is most likely, these sounds are used in games. As an interesting aside, where you see a machine code call made using RANDOMIZE and random numbers are generated in the program, you may like to use RANDOMIZE (Ø * USR nnnnn) (effectively RANDOMIZE Ø) to get around this. However, back to the point. Here are some DATA statements for you to play with:

```
 98 REM   BOMB FALLING
 99 REM   SOUNDS DATA LIST
100 DATA 6,1,197,33,0,0,17,1
110 DATA 0,229,205,181,3,1,20,0
120 DATA 17,0,12,225,198,0,237
130 DATA 74,229,198,0,237,82
140 DATA 225,56,230,193,16,223
150 DATA 201,END
```

```
 98 REM   PHASOR FIRE
 99 REM   SOUNDS DATA LIST
100 DATA 6,1,197,33,0,0,17,1
110 DATA 0,229,205,181,3,1,1,0
120 DATA 17,100,1,225,198,0,237
130 DATA 74,229,198,0,237,82
140 DATA 225,56,230,193,16,223
150 DATA 201,END
```

```
 98 REM   REPEATED PHASOR FIRE
 99 REM   SOUNDS DATA LIST
100 DATA 6,10,197,33,0,0,17,1
110 DATA 0,229,205,181,3,1,1,0
120 DATA 17,100,1,225,198,0,237
130 DATA 74,229,198,0,237,82
140 DATA 225,56,230,193,16,223
150 DATA 201,END
```

```
 98 REM   RASPBERRY
 99 REM   SOUNDS DATA LIST
100 DATA 6,1,197,33,0,10,17,1
110 DATA 0,229,205,181,3,1,100,
0
120 DATA 17,0,30,225,198,0,237
130 DATA 74,229,198,0,237,82
140 DATA 225,56,230,193,16,223
150 DATA 201,END

 98 REM ALIEN MACHINERY OR UFO
 99 REM   SOUNDS DATA LIST
100 DATA 6,20,197,33,0,4,17,1
110 DATA 0,229,205,181,3,1,50,0
120 DATA 17,0,6,225,198,0,237
130 DATA 74,229,198,0,237,82
140 DATA 225,56,230,193,16,223
150 DATA 201,END

 98 REM MYSTERIOUS SOUNDS
 99 REM   SOUNDS DATA LIST
100 DATA 6,5,197,33,0,10,17,10
110 DATA 0,229,205,181,3,1,0,2
120 DATA 17,0,19,225,198,0,237
130 DATA 74,229,198,0,237,82
140 DATA 225,56,230,193,16,223
150 DATA 201,END

 98 REM   ALARM
 99 REM   SOUNDS DATA LIST
100 DATA 6,10,197,33,0,0,17,100
110 DATA 0,229,205,181,3,1,0,1
120 DATA 17,0,3,225,198,0,237
130 DATA 74,229,198,0,237,82
140 DATA 225,56,230,193,16,223
150 DATA 201,END

 98 REM    LASER  BEAM
 99 REM   SOUNDS DATA LIST
100 DATA 6,25,197,33,0,0,17,6
110 DATA 0,229,205,181,3,1,50,0
120 DATA 17,0,1,225,198,0,237
130 DATA 74,229,198,0,237,82
140 DATA 225,56,230,193,16,223
150 DATA 201,END

 98 REM DRUM SYNTH-TYPE 'PEEOW'
 99 REM   SOUNDS DATA LIST
100 DATA 6,1,197,33,0,0,17,5
110 DATA 0,229,205,181,3,1,1,0
120 DATA 17,0,1,225,198,0,237
130 DATA 74,229,198,0,237,82
140 DATA 225,56,230,193,16,223
150 DATA 201,END

 98 REM BIRD(OR SEEING STARS!!)
 99 REM   SOUNDS DATA LIST
100 DATA 6,14,197,33,0,0,17,40
110 DATA 0,229,205,181,3,1,25,0
```

```
120 DATA 17,240,0,225,198,0,237
130 DATA 74,229,198,0,237,82
140 DATA 225,56,230,193,16,223
150 DATA 201,END
```

After being used to the Spectrum's polite little BEEP in BASIC, these sounds are quite a happy surprise. The one thing the routine given can't do is give a rising pitch tone. Normally this won't make much difference because the sound repeats itself so quickly it doesn't matter. But if you want specifically a note of rising pitch, here's a slightly different version of the same routine to do this.

HEXA-DECIMAL	DECIMAL	ASSEMBLER	REMARKS
06 01	6,1	LD B,1	; repeats counter
C5	197	REPT ;PUSH BC	; save counter
21 E8 03	33,232,3	LD HL,1000	; start pitch
11 01 00	17,1,0	LOOP ;LD DE,1	; duration of sound
E5	229	PUSH HL	; save pitch
CD B5 03	205,181,3	CALL 949	; call BEEP in ROM
E1	225	POP HL	; recover pitch
01 01 00	1,1,0	LD BC,1	; change of pitch
C6 00	198,0	ADD A,0	; reset carry flag
ED 42	237,66	SBC HL,BC	; new pitch
30 EF	48,239	JR NC,LOOP (JR NC, – 17)	; more of this sound?
C1	193	POP BC	; nope. Recover repeat counter
10 E8	16,232	DJ NZ,REPT (DJ NZ, – 24)	; more repeats?
C9	201	RET	; back to BASIC

This routine is only 27 bytes long and lacks the "limit pitch" feature, meaning that whatever pitch the sound starts off with, it always ends up at its highest. You could incorporate it into a DATA statement like the others, although there'd be a few bytes wasted which you'd have to change the CLEAR statement in line 10 of the loader program if you were worried about that. However, leaving it as it is does have the advantage of being standard in that you can swap any of them about as you please. If you wanted this shorter routine to be the same length, you could pad it out with zeros (NOP) before END in the DATA.

```
 98 REM    ASCENDING TONE
 99 REM   SOUNDS DATA LIST
100 DATA 6,1,197,33,232,3,17,1
110 DATA 0,229,205,181,3,225,1
120 DATA 1,0,198,0,237,66,48
130 DATA 239,193,16,232,201,END

 98 REM FASTER ASCENDING TONE
 99 REM   SOUNDS DATA LIST
100 DATA 6,1,197,33,232,2,17,1
110 DATA 0,229,205,181,3,225,1
```

```
120 DATA 10,0,198,0,237,66,48
130 DATA 239,193,16,232,201,END
```

Now the next question will be how to go about adding these sounds to our programs. It'll involve a bit of sweat and tears for those with no experience of machine code, I'm afraid. There are three main options:

(i) Incorporate a loader program and the DATA statements into your program. When the program is RUN (with the SAVE LINE facility for convenience, then you need only set up the machine once. Subsequent RUNs will only run the program and not set up the machine code again,) this might take the form:

9900 REM loader program

.....

9950 REM DATA statements here

.....

9999 GOTO 1

This program would have been saved with SAVE "program" LINE 9900

(ii) Set up the machine code beforehand and save it on tape using the SAVE CODE facility for subsequent reloading by the program concerned. A time consuming exercise.

(iii) Store it all ready made in a REM statement in the program. This would be saved on tape in its own right as a line to be merged into your programs that require these sounds. The whole lot could then be saved together so that everything was ready to go on loading — no annoying delay while the machine code side of things are set up. This program will allow you to do this for the twelve sounds we've discussed. Remember that we said each routine was 36 bytes long, and since there are 12 routines we'll need a REM statement with at least 12 * 36 or 432 characters after the word REM in that one line. That's a lot of typing, so make sure your finger is fit and ready for the ordeal.

Line 10 to 60 comprise the loader program. The rest of the program is DATA statements holding the machine code for the various sounds. Obviously, you could omit the REM statements as they only show where the DATA for each sound begins. Note that there is only a variable END at the very end of the list — in case you were expecting them to terminate every sound from earlier on. There's a lot of typing to do and it's important that it all gets done properly as nothing will work after any mistake. As soon as you type it all in, save it on tape in case any errors cause you to lose it all!

```
  1 REM ..........................
.................................
.................................
.................................
.................................
.................................
.................................
.................................
.................................
.................................
.................................
.................................
.................................
.................................
.................................
.....................
  10 LET ADDRESS=PEEK 23635+256*
PEEK 23636+5: LET END=1000
  20 READ BYTE
  30 IF BYTE>255 THEN STOP
  40 POKE ADDRESS,BYTE
  50 LET ADDRESS=ADDRESS+1
  60 GO TO 20
 100 REM BOMB FALLING
 110 DATA 6,1,197,33,0,0,17,1
 120 DATA 0,229,205,161,3,1,20,0
 130 DATA 17,0,12,225,198,0,237
 140 DATA 74,229,198,0,237,82
 150 DATA 225,56,230,193,16,223
 160 DATA 201
 170 REM PHASOR FIRE
 180 DATA 6,1,197,33,0,0,17,1
 190 DATA 0,229,205,161,3,1,1,0
 200 DATA 17,100,1,225,198,0,237
 210 DATA 74,229,198,0,237,82
 220 DATA 225,56,230,193,16,223
 230 DATA 201
 240 REM REPEATED PHASOR FIRE
 250 DATA 6,10,197,33,0,0,17,1
 260 DATA 0,229,205,161,3,1,1,0
 270 DATA 17,100,1,225,198,0,237
 280 DATA 74,229,198,0,237,82
 290 DATA 225,56,230,193,16,223
 300 DATA 201
```

```
310 REM RASPBERRY
320 DATA 6,1,197,33,0,10,17,1
330 DATA 0,229,205,161,3,1,100,
0
340 DATA 17,0,30,225,198,0,237
350 DATA 74,229,198,0,237,82
360 DATA 225,56,230,193,16,223
370 DATA 201
380 REM ALIEN MACHINERY OR UFO.
390 DATA 6,20,197,33,0,4,17,1
400 DATA 0,229,205,161,3,1,50,0
410 DATA 17,0,6,225,198,0,237
420 DATA 74,229,198,0,237,82
430 DATA 225,56,230,193,16,223
440 DATA 201
450 REM MYSTERIOUS SOUNDS
460 DATA 6,5,197,33,0,10,17,10
470 DATA 0,229,205,161,3,1,0,2
480 DATA 17,0,19,225,198,0,237
490 DATA 74,229,198,0,237,82
500 DATA 225,56,230,193,16,223
510 DATA 201
520 REM ALARM
530 DATA 6,10,197,33,0,0,17,100
540 DATA 0,229,205,161,3,1,0,1
550 DATA 17,0,3,225,198,0,237
560 DATA 74,229,198,0,237,82
570 DATA 225,56,230,193,16,223
580 DATA 201
590 REM LASER BEAM
600 DATA 6,25,197,33,0,0,17,6
610 DATA 0,229,205,161,3,1,50,0
620 DATA 17,0,1,225,198,0,237
630 DATA 74,229,198,0,237,82
640 DATA 225,56,230,193,16,223
650 DATA 201
660 REM DRUM SYNTH-TYPE 'PEEOW'
670 DATA 6,1,197,33,0,0,17,5
680 DATA 0,229,205,161,3,1,1,0
690 DATA 17,0,1,225,198,0,237
700 DATA 74,229,198,0,237,82
710 DATA 225,56,230,193,16,223
720 DATA 201
730 REM BIRD (OR SEEING STARS!)
740 DATA 6,14,197,33,0,0,17,40
750 DATA 0,229,205,161,3,1,25,0
760 DATA 17,240,0,225,198,0,237
770 DATA 74,229,198,0,237,82
780 DATA 225,56,230,193,16,223
790 DATA 201
800 REM ASCENDING TONE
810 DATA 6,1,197,33,232,3,17,1
820 DATA 0,229,205,161,3,225,1
830 DATA 1,0,198,0,237,66,48
840 DATA 239,193,16,232,201
850 DATA 0,0,0,0,0,0,0,0,0,0
860 REM FASTER ASCENDING TONE
870 DATA 6,1,197,33,232,2,17,1
880 DATA 0,229,205,161,3,225,1
890 DATA 10,0,198,0,237,66,48
```

```
900 DATA 239,193,16,232,201
910 DATA END
```

After SAVEing it all on tape, RUN the program to set up the machine code, which will take a few seconds and the program will stop with report 9 STOP statement, 30:2. As a quick check for everything being OK, try the command PRINT ADDRESS, which should give 24183. This only works if you have no expansion module, etc., plugged in as the program then will have started at a different address. Ultimately, the only test is to RUN the machine code — this is why I advised you to SAVE it all on tape beforehand. Enter this short test program before you start to delete any lines:

```
  10 LET ADDRESS=PEEK 23635+256*
PEEK 23636+5
  11 FOR A=ADDRESS TO ADDRESS+42
3 STEP 36
  12 RANDOMIZE USR A
  13 PAUSE 20
  14 NEXT A: STOP
```

If all went well, you should have heard the twelve sounds with a slight pause between each one and the program would have stopped normally in line 14. If all's well and good you can go ahead and delete every line except the line 1 REM and SAVE the line 1 REM on tape in its own right. This is the line you'll add to your future games programs. Now as to how to call each sound as you need them. Those with no expansion module don't know how lucky they are — they can just call the appropriate address since the REM will be fixed in memory. This table tells you where to call:

```
310 REM RASPBERRY
320 DATA 6,1,197,33,0,10,17,1
330 DATA 0,229,205,161,3,1,100,
0
340 DATA 17,0,30,225,198,0,237
350 DATA 74,229,198,0,237,82
360 DATA 225,56,230,193,16,223
370 DATA 201
380 REM ALIEN MACHINERY OR UFO.
390 DATA 6,20,197,33,0,4,17,1
400 DATA 0,229,205,161,3,1,50,0
410 DATA 17,0,6,225,198,0,237
420 DATA 74,229,198,0,237,82
430 DATA 225,56,230,193,16,223
440 DATA 201
450 REM MYSTERIOUS SOUNDS
460 DATA 6,5,197,33,0,10,17,10
470 DATA 0,229,205,161,3,1,0,2
480 DATA 17,0,19,225,198,0,237
490 DATA 74,229,198,0,237,82
500 DATA 225,56,230,193,16,223
510 DATA 201
520 REM ALARM
530 DATA 6,10,197,33,0,0,17,100
540 DATA 0,229,205,161,3,1,0,1
550 DATA 17,0,3,225,198,0,237
560 DATA 74,229,198,0,237,82
570 DATA 225,56,230,193,16,223
580 DATA 201
590 REM LASER BEAM
600 DATA 6,25,197,33,0,0,17,6
610 DATA 0,229,205,161,3,1,50,0
620 DATA 17,0,1,225,198,0,237
630 DATA 74,229,198,0,237,82
640 DATA 225,56,230,193,16,223
650 DATA 201
660 REM DRUM SYNTH-TYPE 'PEEOW'
670 DATA 6,1,197,33,0,0,17,5
680 DATA 0,229,205,161,3,1,1,0
690 DATA 17,0,1,225,198,0,237
700 DATA 74,229,198,0,237,82
710 DATA 225,56,230,193,16,223
720 DATA 201
730 REM BIRD (OR SEEING STARS!)
740 DATA 6,14,197,33,0,0,17,40
750 DATA 0,229,205,161,3,1,25,0
760 DATA 17,240,0,225,198,0,237
770 DATA 74,229,198,0,237,82
780 DATA 225,56,230,193,16,223
790 DATA 201
800 REM ASCENDING TONE
810 DATA 6,1,197,33,232,3,17,1
820 DATA 0,229,205,161,3,225,1
830 DATA 1,0,198,0,237,66,46
840 DATA 239,193,16,232,201
850 DATA 0,0,0,0,0,0,0,0,0,0
860 REM FASTER ASCENDING TONE
870 DATA 6,1,197,33,232,2,17,1
880 DATA 0,229,205,161,3,225,1
890 DATA 10,0,198,0,237,66,46
```

```
900 DATA 239,193,16,232,201
910 DATA END
```

After SAVEing it all on tape, RUN the program to set up the machine code, which will take a few seconds and the program will stop with report 9 STOP statement, 30:2. As a quick check for everything being OK, try the command PRINT ADDRESS, which should give 24183. This only works if you have no expansion module, etc., plugged in as the program then will have started at a different address. Ultimately, the only test is to RUN the machine code — this is why I advised you to SAVE it all on tape beforehand. Enter this short test program before you start to delete any lines:

```
  10 LET ADDRESS=PEEK 23635+256*
PEEK 23636+5
  11 FOR A=ADDRESS TO ADDRESS+42
3 STEP 36
  12 RANDOMIZE USR A
  13 PAUSE 20
  14 NEXT A: STOP
```

If all went well, you should have heard the twelve sounds with a slight pause between each one and the program would have stopped normally in line 14. If all's well and good you can go ahead and delete every line except the line 1 REM and SAVE the line 1 REM on tape in its own right. This is the line you'll add to your future games programs. Now as to how to call each sound as you need them. Those with no expansion module don't know how lucky they are — they can just call the appropriate address since the REM will be fixed in memory. This table tells you where to call:

SOUND NUMBER	USR ADDRESS	SOUND
1	23760	Bomb falling
2	23796	Phasor fire
3	23832	Repeated phasor fire
4	23868	Raspberry
5	23904	Alien machinery/UFO
6	23940	Mysterious sounds
7	23976	Alarm
8	24012	Laser beam
9	24048	Drum – synthesiser "peeow" sound
10	24084	Bird (or seeing stars!!)
11	24120	Ascending tone
12	24156	Faster ascending tone

Calling by those addresses will probably not work for users whose Spectrums have microdrive maps and so on below the program area. So I suggest you call the appropriate routine by PEEKing 23635/6 and adding the appropriate number of bytes. This would be very tedious so I suggest you include an FN call to calculate it for you:

```
DEF FN U(N) = PEEK 23635 + 256*PEEK 23636 + 5 +
36*(N - 1)
```

Include that at line 2 and SAVE it with the REM.

Any time you want one of those sounds (e.g. the sixth sound):

```
LET U = USR FN U (6)
```

Seeing as the sounds will be used in games, it is better to assign the number returned from the USR call to a variable than use RANDOMIZE USR, since this will affect random numbers used. Here is an example to show how to use the USR FN:

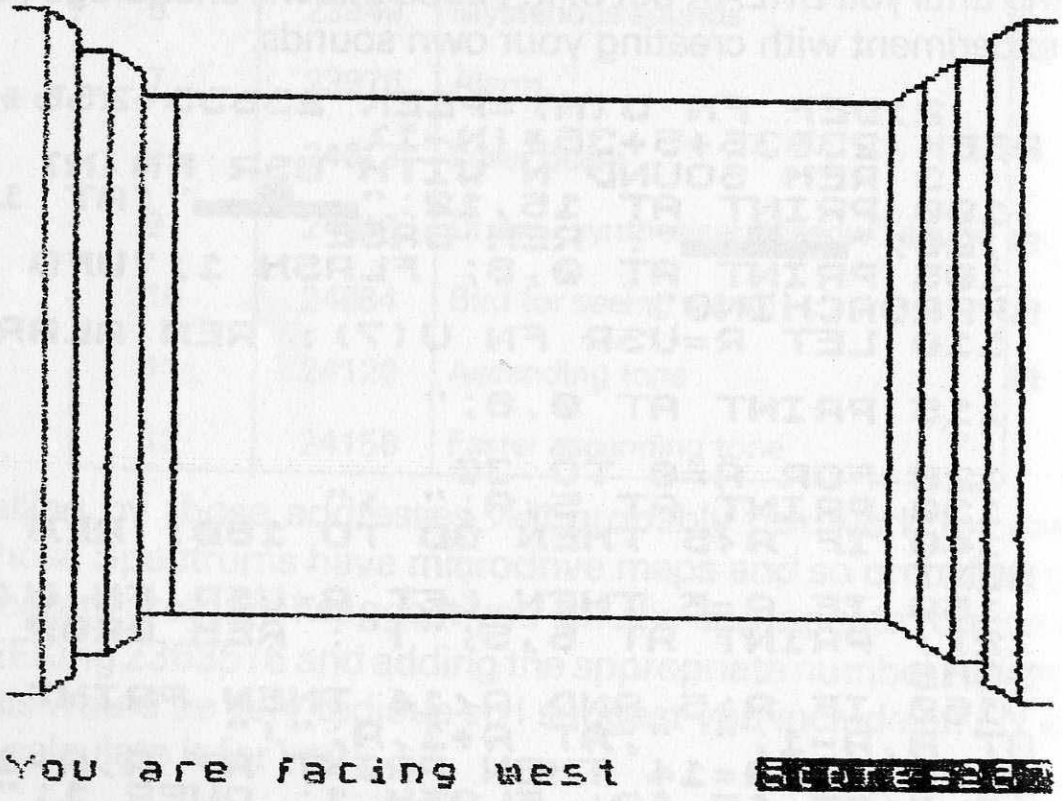
```
    2>DEF FN U(N)=PEEK 23635+256*
PEEK 23636+5+36*(N-1)
    3 REM SOUND N WITH USR FN(N)
   10 FOR A=1 TO 12
   20 LET U=USR FN U(A)
   30 NEXT A
```

Here is an example graphics program to show how to use the sounds in a program. An alarm sounds over the two missile bases that a UFO is approaching. On its first flypast it destroys a base by dropping a bomb on it and then, on its second flypast, the second base shoots it — all rather violent stuff and a bit overdone, but a useful demonstration of the sounds. It keeps going until you BREAK out of it. I hope this will encourage you to experiment with creating your own sounds.

```
    2>DEF FN U(N)=PEEK 23635+256*
PEEK 23636+5+36*(N-1)
    3 REM SOUND N WITH USR FN(N)
  100 PRINT AT 15,12;"▄▄▄▄▄▄";AT 1
5,20;"▄▄▄▄▄▄": REM BASE
  105 PRINT AT 0,8; FLASH 1;"UFO
APPROACHING"
  110 LET R=USR FN U(7): REM ALAR
M
  115 PRINT AT 0,8;"
   "
  120 FOR A=0 TO 30
  130 PRINT AT 5,A;" >"
  140 IF A<5 THEN GO TO 180: REM
NEXT
  150 IF A=5 THEN LET R=USR FN U(
12): PRINT AT 6,5;"!": REM DROP
BOMB
  160 IF A>5 AND A<14 THEN PRINT
AT A,A-1;" ";AT A+1,A;"!"
  170 IF A=14 THEN PRINT AT A,A-1
;" ";AT 15,12; FLASH 1; OVER 1;"
        ": LET R=USR FN U(1): PRINT
 AT 15,12;"        "
  180 NEXT A
  185 PRINT AT 5,31;" "
  186 PRINT AT 0,8; FLASH 1;"UFO
APPROACHING"
  187 LET R=USR FN U(7): REM ALAR
M
  188 PRINT AT 0,8;"
   "
  190 FOR A=0 TO 21
  200 PRINT AT 5,A;" >"
  210 NEXT A
  220 PLOT 179,56: DRAW 0,75
  225 LET R=USR FN U(8)
  230 PLOT OVER 1;179,56: DRAW OV
ER 1;0,75
  240 PRINT AT 5,22; FLASH 1;">"
  250 LET R=USR FN U(4)
  255 PAUSE 50
  260 CLS : GO TO 100
```

3D MAZE (16K)

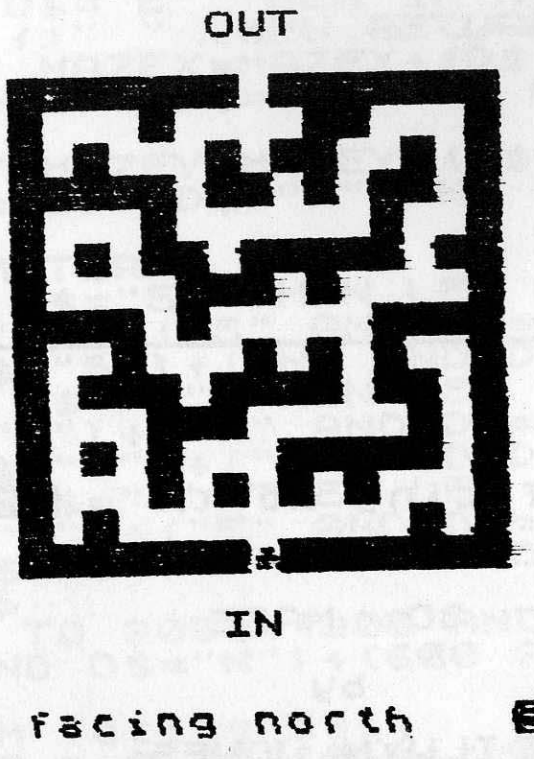
Fancy taking a walk in a maze where you can see the walls and corridors in glorious breathtaking 3D effect? If you're getting lost in an endless corridor, consult your computer for some help and a diagram at the expense of losing 20 points out of the 300 you start with. The object, of course, being to get out of the maze successfully with as high a score as possible, your score decreases every time you try to move and by 20 points every time you ask for help. And once you crack the maze, you can change the design for another one.



Anyway, begin by typing in the listing which should run on a 16K machine. Remember that there is no difference between upper case and lower case variable names, so that MAZEY is the same as mazey, for example. Follow the line numbers exactly as there are several calculated GOSUBs and GOTOs. Where you see references to D$, any reference to "N" for North, "S" for south, "E" for east, or "W" for west must be made in UPPER CASE letters as the program uses this variable to keep track of which direction you are facing in and only checks for the upper case letters. There are also several multi-statement IF . . . THEN . . . lines in the program. These are used when more than an outcome for a certain condition is needed but having a separate subroutine is not really needed. In line

8Ø15 there are four apostrophes in the PRINT statement. In line 851Ø and 8515 the symbols after PRINT are the hash (noughts and crosses board lookalike) on the 3 key. In the maze DATA the characters I used were graphics SHIFT 8 squares, but these could be any character you prefer as the program only checks for the presence of a SPACE to denote an open corridor, and any other character denotes a wall in the maze.

Once all the program has been entered, save it on tape and verify it. Then when you are ready, run the program. The display will change to white on blue with a black BORDER. A two note bleep will sound once the maze and all DATA has been set up. The message "Press any key when you are ready" will appear and once you press any of the keys on the keyboard you will be presented with a diagram of the maze for about two seconds with the entry marked "IN" and the exit marked "OUT", and a message at the bottom of the screen shows what direction you are facing in, and the starting score. Your position is indicated by an asterisk at the entry point at this time.

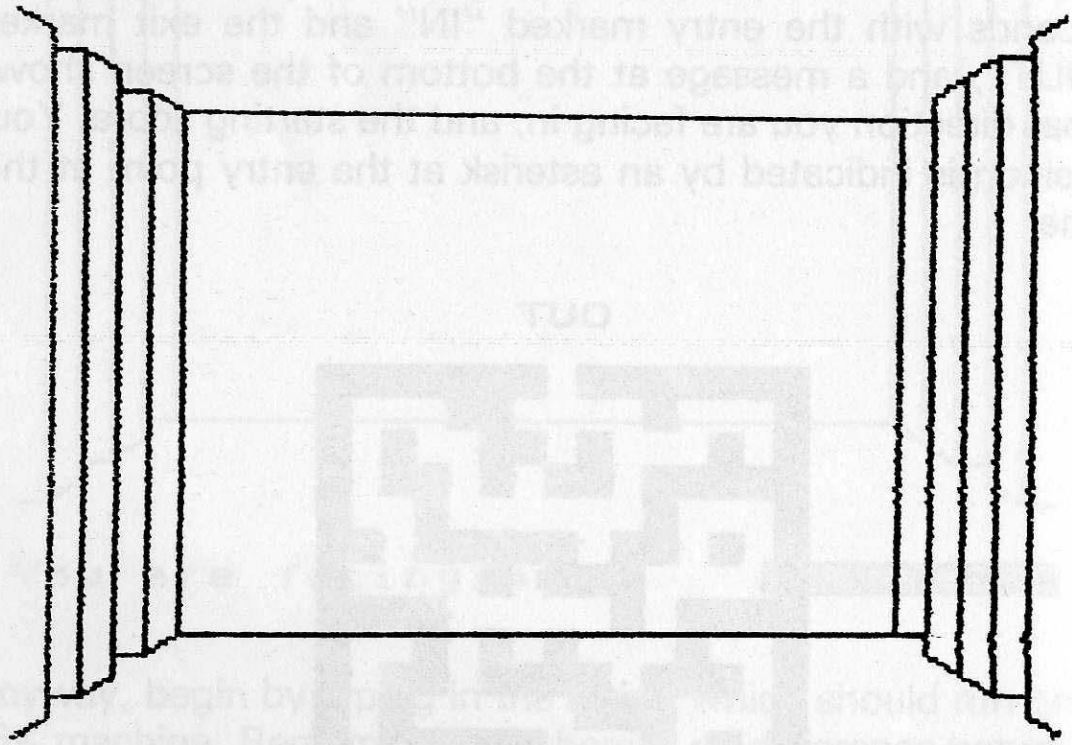


The screen will now clear and you are alone in the maze. Ahead of you is a 3D view of where you face in the maze. I'd better tell you what the controls at your disposal are:

5 : Turn to your left (rotate 90 degrees anticlockwise).

6 : Turn around backwards (rotate through 180 degrees).
7 : Move forward (actually one square in the diagram).
8 : Turn right (rotate 90 degrees clockwise).
h or H : Help!!!!!
q or Q : I Quit. . . I've had enough. . . get me out of here. . .

The last option really should be the *last* option as all it does is stop the program. Use this rather than BREAK in case you leave some RETURN addresses on the stack and clog up your memory. You could use the Q option to suspend the program while you're answering the phone or something, then restart with CONTINUE, although you won't get a direction indication until you actually make a move since the direction prompt and score use the lower screen and this would have been cleared by the STOP report.

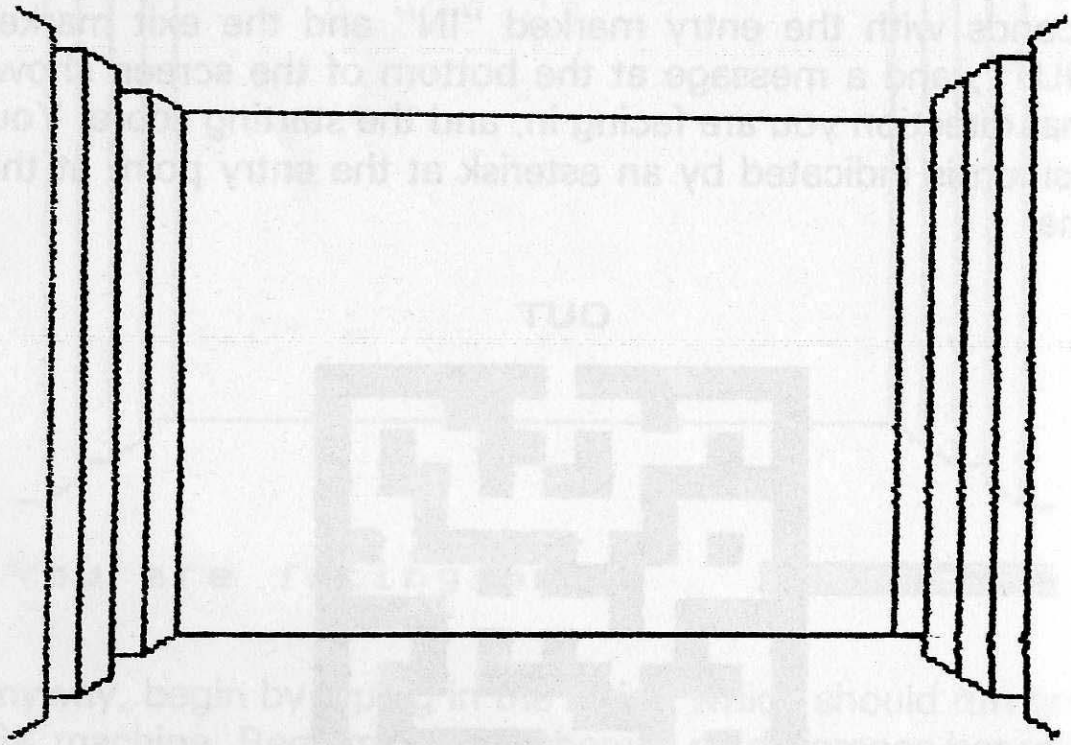



```
    1 REM    3D   MAZE
    2
    3 REM        by
    4
    5 REM DILWYN JONES
    6
   10 BRIGHT 0: FLASH 0: INVERSE
0: OVER 0
   20 INK 7: PAPER 1: BORDER 0: C
LS
   30 GO SUB 9000
  100 REM KEYBOARD
```

```
 110 LET K$=INKEY$
 112 IF K$="H" OR K$="h" THEN GO
 SUB 8000: GO SUB 8500
 113 IF k$="q" OR k$="Q" THEN ST
OP
 115 IF (K$<"5" OR K$>"8") THEN
GO TO 110
 117 LET score=score-1
 120 IF K$="5" OR K$="6" OR K$="
8" THEN CLS : GO SUB 1200: GO SU
B 8500
 130 IF K$="7" THEN GO SUB 1000:
 GO SUB 8500
 140 IF mazey=exity AND mazex=ex
itx THEN PRINT INVERSE 1;AT 1,3;
" OUT WITH A SCORE OF ";SCORE: S
TOP
 200 GO TO 100
 230 STOP
1000 REM MOVE FORWARD
1005 IF (D$="N" AND MAZEY=1) OR
(D$="S" AND MAZEY=15) OR (D$="E"
 AND MAZEX=15) OR (D$="W" AND MA
ZEX=1) THEN RETURN
1010 IF D$="N" THEN IF M$(MAZEY-
1,MAZEX)<>" " THEN RETURN
1020 IF D$="S" THEN IF M$(MAZEY+
1,MAZEX)<>" " THEN RETURN
1030 IF D$="W" THEN IF M$(MAZEY,
MAZEX-1)<>" " THEN RETURN
1040 IF D$="E" THEN IF M$(MAZEY,
MAZEX+1)<>" " THEN RETURN
1050 LET MAZEX=MAZEX+(D$="E" AND
 MAZEX<=15)-(D$="W" AND MAZEX>=1
)
1060 LET MAZEY=MAZEY+(D$="S" AND
 MAZEY<=15)-(D$="N" AND MAZEY>=1
)
1200 REM TURN
1210 IF K$="5" THEN LET D$=("W"
AND D$="N")+("S" AND D$="W")+("E
" AND D$="S")+("N" AND D$="E")
1220 IF K$="8" THEN LET D$=("E"
AND D$="N")+("N" AND D$="W")+("W
" AND D$="S")+("S" AND D$="E")
1230 IF K$="6" THEN LET D$=("S"
AND D$="N")+("E" AND D$="W")+("W
" AND D$="S")+("W" AND D$="E")
1235 CLS
1240 GO TO 2000+(200 AND D$="W")
+(400 AND D$="N")+(600 AND D$="S
")
2000 REM facing east
2010 FOR m=0 TO 15-mazex
2020 LET x=6*m
2025 IF MAZEY=15 THEN GO SUB 700
0: GO SUB 4000+(2000 AND M$(MAZE
Y-1,MAZEX-M)=" "): GO TO 2070
2026 IF MAZEY=1 THEN GO SUB 6000
: GO SUB 5000+(2000 AND M$(MAZEY
+1,MAZEX+M)=" "): GO TO 2070
```

6 : Turn around backwards (rotate through 180 degrees).
7 : Move forward (actually one square in the diagram).
8 : Turn right (rotate 90 degrees clockwise).
h or H : Help!!!!!
q or Q : I Quit. . . I've had enough. . . get me out of here. . .

The last option really should be the *last* option as all it does is stop the program. Use this rather than BREAK in case you leave some RETURN addresses on the stack and clog up your memory. You could use the Q option to suspend the program while you're answering the phone or something, then restart with CONTINUE, although you won't get a direction indication until you actually make a move since the direction prompt and score use the lower screen and this would have been cleared by the STOP report.




```
    1 REM    3D   MAZE
    2
    3 REM        by
    4
    5 REM DILWYN JONES
    6
   10 BRIGHT 0: FLASH 0: INVERSE
0: OVER 0
   20 INK 7: PAPER 1: BORDER 0: C
LS
   30 GO SUB 9000
  100 REM KEYBOARD
```

```
 110 LET K$=INKEY$
 112 IF K$="H" OR K$="h" THEN GO
 SUB 8000: GO SUB 8500
 113 IF k$="q" OR k$="Q" THEN ST
OP
 115 IF (K$<"5" OR K$>"8") THEN
GO TO 110
 117 LET score=score-1
 120 IF K$="5" OR K$="6" OR K$="
8" THEN CLS : GO SUB 1200: GO SU
B 8500
 130 IF K$="7" THEN GO SUB 1000:
 GO SUB 8500
 140 IF mazey=exity AND mazex=ex
itx THEN PRINT INVERSE 1;AT 1,3;
" OUT WITH A SCORE OF ";SCORE: S
TOP
 200 GO TO 100
 230 STOP
1000 REM MOVE FORWARD
1005 IF (D$="N" AND MAZEY=1) OR
 (D$="S" AND MAZEY=15) OR (D$="E"
 AND MAZEX=15) OR (D$="W" AND MA
ZEX=1) THEN RETURN
1010 IF D$="N" THEN IF M$(MAZEY-
1,MAZEX)<>" " THEN RETURN
1020 IF D$="S" THEN IF M$(MAZEY+
1,MAZEX)<>" " THEN RETURN
1030 IF D$="W" THEN IF M$(MAZEY,
MAZEX-1)<>" " THEN RETURN
1040 IF D$="E" THEN IF M$(MAZEY,
MAZEX+1)<>" " THEN RETURN
1050 LET MAZEX=MAZEX+(D$="E" AND
 MAZEX<=15)-(D$="W" AND MAZEX>=1
)
1060 LET MAZEY=MAZEY+(D$="S" AND
 MAZEY<=15)-(D$="N" AND MAZEY>=1
)
1200 REM TURN
1210 IF K$="5" THEN LET D$=("W"
AND D$="N")+("S" AND D$="W")+("E
" AND D$="S")+("N" AND D$="E")
1220 IF K$="8" THEN LET D$=("E"
AND D$="N")+("N" AND D$="W")+("W
" AND D$="S")+("S" AND D$="E")
1230 IF K$="6" THEN LET D$=("S"
AND D$="N")+("E" AND D$="W")+("W
" AND D$="S")+("W" AND D$="E")
1235 CLS
1240 GO TO 2000+(200 AND D$="W")
+(400 AND D$="N")+(600 AND D$="S
")
2000 REM facing east
2010 FOR m=0 TO 15-mazex
2020 LET x=6*m
2025 IF MAZEY=15 THEN GO SUB 700
0: GO SUB 4000+(2000 AND M$(MAZE
Y-1,MAZEX-M)=" "): GO TO 2070
2026 IF MAZEY=1 THEN GO SUB 6000
: GO SUB 5000+(2000 AND M$(MAZEY
+1,MAZEX+M)=" "): GO TO 2070
```

```
2030 GO SUB 4000+(2000 AND m$(ma
zey-1,mazex+m)=" ")
2050 GO SUB 5000+(2000 AND m$(ma
zey+1,mazex+m)=" ")
2070 IF mazex+m+1<=15 THEN IF m$
(mazey,mazex+m+1)<>" " THEN GO T
O 3500
2080 NEXT m
2090 RETURN
2200 REM facing west
2210 FOR m=0 TO mazex-1
2220 LET x=8*m
2225 IF MAZEY=15 THEN GO SUB 600
0: GO SUB 5000+(2000 AND M$(MAZE
Y-1,MAZEX-M)=" "): GO TO 2270
2226 IF MAZEY=1 THEN GO SUB 7000
: GO SUB 4000+(2000 AND M$(MAZEY
+1,MAZEX-M)=" "): GO TO 2270
2230 GO SUB 4000+(2000 AND m$(ma
zey+1,mazex-m)=" ")
2250 GO SUB 5000+(2000 AND m$(ma
zey-1,mazex-m)=" ")
2270 IF mazex-m-1>=1 THEN IF m$(
mazey,mazex-m-1)<>" " THEN GO TO
 3500
2280 NEXT m
2290 RETURN
2400 REM face north
2410 FOR m=0 TO mazey-1
2420 LET x=m*8
2425 IF MAZEX=15 THEN GO SUB 700
0: GO SUB 4000+(2000 AND M$(MAZE
Y-m,MAZEX-1)=" "): GO TO 2470
2426 IF MAZEX=1 THEN GO SUB 6000
: GO SUB 5000+(2000 AND M$(MAZEY
-m,MAZEX+1)=" "): GO TO 2470
2430 GO SUB 4000+(2000 AND m$(ma
zey-m,mazex-1)=" ")
2450 GO SUB 5000+(2000 AND m$(ma
zey-m,mazex+1)=" ")
2470 IF mazey-m-1>=1 THEN IF M$(
MAZEY-M-1,MAZEX)<>" " THEN GO TO
 3500
2480 NEXT M
2490 RETURN
2600 REM facing south
2610 FOR m=0 TO 15-mazey
2620 LET x=8*m
2625 IF MAZEX=15 THEN GO SUB 600
0: GO SUB 5000+(2000 AND M$(MAZE
Y+m,MAZEX-1)=" "): GO TO 2670
2626 IF MAZEX=1 THEN GO SUB 7000
: GO SUB 4000+(2000 AND M$(MAZEY
+m,MAZEX+1)=" "): GO TO 2670
2630 GO SUB 4000+(2000 AND m$(ma
zey+m,mazex+1)=" ")
2650 GO SUB 5000+(2000 AND m$(ma
zey+m,mazex-1)=" ")
2670 IF mazey+m+1<=15 THEN IF m$
(mazey+m+1,mazex)<>" " THEN GO T
O 3500
```

```
2680 NEXT m
2690 RETURN
3500 REM wall ahead
3510 LET x=x+8
3520 PLOT x,x*5/8: DRAW 255-(2*x
),0
3530 PLOT x,x*5/8+175-(10*x/8):
DRAW 255-(2*x),0
3540 RETURN
4000 REM DRAW BLOCKED OFF WALL
ON LEFT
4010 PLOT X,X*5/8
4020 DRAW 8,5
4030 DRAW 0,165-(10*X/8)
4040 DRAW -8,5
4050 RETURN
5000 REM DRAW BLOCKED OFF WALL
ON RIGHT
5010 PLOT 255-X,X*5/8
5020 DRAW -8,5
5030 DRAW 0,165-(5*X/4)
5040 DRAW 8,5
5050 RETURN
6000 REM DRAW OPEN CORRIDOR ON
LEFT
6010 PLOT X,X*5/8+5
6020 DRAW 8,0
6030 DRAW 0,165-(5*X/4)
6040 DRAW -8,0
6050 RETURN
7000 REM DRAW OPEN CORRIDOR ON
RIGHT
7010 PLOT 255-X,X*5/8+5
7020 DRAW -8,0
7030 DRAW 0,165-(5*X/4)
7040 DRAW 8,0
7050 RETURN
8000 REM HELP
8010 CLS
8015 PRINT ....
8020 FOR A=1 TO 15
8030 PRINT TAB 8;M$(A)
8040 NEXT A
8050 PRINT AT MAZEY+3,MAZEX+7;"*
"
8051 IF EXITY=1 OR EXITY=15 THEN
 PRINT AT 2+(18 AND EXITY=15),EX
ITX+6;"OUT"
8052 IF EXITY>1 AND EXITY<15 THE
N PRINT AT EXITY+3,4+(20 AND EXI
TX=15);"OUT"
8053 IF ENTRYY=1 OR ENTRYY=15 TH
EN PRINT AT 2+(18 AND ENTRYY=15)
,ENTRYX+6;"IN"
8054 IF ENTRYY>1 AND ENTRYY<15 T
HEN PRINT AT ENTRYY+3,5+(19 AND
ENTRYX=15);"IN"
8059 LET score=score-20: GO SUB
8500
8060 PAUSE 100: CLS
8070 GO SUB 2000+(200 AND D$="W"
```

```
) +(400 AND D$="N")+(600 AND D$="
S")
8090 RETURN
8500 REM WHICH DIRECTION?
8505 IF score<0 THEN LET score=0
8510 PRINT #1;AT 1,0;"You are fa
cing ";"north" AND D$="N";"sout
h" AND D$="S";"east " AND D$="E"
;"west " AND D$="W"
8515 PRINT #1;AT 1,26-LEN STR$ s
core; INVERSE 1;"Score=";score
8520 RETURN
9000 REM INITIALISE
9010 REM DATA FOR MAZE
9020 DATA "
9030 DATA "
9040 DATA "
9050 DATA "
9060 DATA "
9070 DATA "
9080 DATA "
9090 DATA "
9100 DATA "
9110 DATA "
9120 DATA "
9130 DATA "
9140 DATA "
9150 DATA "
9160 DATA "
9170 LET ENTRYX=8: LET ENTRYY=15
9180 LET EXITX=8: LET EXITY=1
9190 DIM M$(15,15): REM MAZE
9200 RESTORE 9020
9210 FOR M=1 TO 15
9220 READ M$(M)
9230 NEXT M
9240 LET MAZEX=ENTRYX
9250 LET MAZEY=ENTRYY
9260 LET D$="N"
9261 BEEP .5,0: BEEP .7,12
9262 IF INKEY$<>"" THEN GO TO 92
62
9263 PRINT "Press any key when y
ou are ready"
9264 IF INKEY$="" AND IN 65278=2
55 AND IN 32766=255 THEN GO TO 9
264
9265 LET score=320
9266 GO SUB 8000
9267 GO SUB 6500
9270 GO SUB 2400
9290 GO SUB 6500
9300 RETURN
```

The listing is fairly complex and long. There is a lot of repetition in the drawing routines to avoid a multitude of conditional statements all over the place to cover every eventuality which

although shortening the program might actually slow it down if done. As it stands, you may well be pleasantly surprised to find a BASIC program of this complexity running this fast since there is quite a lot of working out perspective views and drawing them to be done. The drawing is not strictly accurate in that although height changes with distance, the width of the corridor entrances do not change as you go deeper towards the far end of the corridor. This would not only slow down the program to an unacceptable extent, but mean that the furthest wall or side entrance ahead of you would be so thin as to be indistinguishable. As you can see from the example, the method used for drawing gives more than an acceptable 3D effect, where you can see blocked off walls to the left or right and clearly tell the difference between these and open corridors to the left or right or ahead.

Lines 10 and 20 set up the colours, etc., for the display. These are the only global colour commands, so if white text and graphics on a blue background does not agree with you change these statements. The black BORDER is used to frame the view ahead and some do not like this; again, if you change this statement it will provide the BORDER used throughout the program. Line 30 sends the program to the initialisation routine at line 9000. The loop in lines 100 to 200 deals with reading the keyboard and taking appropriate action. Line 112 deals with the "help" option by going to the line 8000 routine (help) and 8500 (direction/score). The help routine deducts the appropriate amount from the score within itself, and the direction/score routine both prints the direction in which you are currently facing and the present score. Line 113 deals with the Quit option. Line 115 ensures that the program goes no further if no key is being pressed or the help option has been taken. This would prevent the score working properly. Line 117 decrements the score by one for every move made except the help option where different action is taken. Line 120 deals with the turning options by first clearing the screen then calling the turning subroutine at line 1200 then the direction/score routine. Line 130 deals with moving forward by calling subroutine 1000 and the direction/score routine. Line 140 deals with checking to see if you've made it through the maze and prints the score and stops the program. At line 1000, you start the move forward subroutine. Line 1005

checks for the limits of the maze. Lines 1010 to 1040 check if there is an obstruction ahead. Note the use of IF...THEN IF...THEN... in place of IF...AND...THEN... since the second condition may cause an error to arise under certain conditions such as turning sideways in the entrance to the maze. The program jumps over these without trying to execute anything that might cause an error. This is a feature of ZX Spectrum BASIC that anything after THEN is completely skipped over to the end of the line unless the condition is true. Lines 1050 and 1060 change the values of the variables MAZEY and MAZEX as appropriate. These two variables are the position down the maze and across the maze respectively. Line 1200 is the start of the turn routine which changes the direction variable D$ as needed. Line 1240 chooses which drawing routine to use depending on your direction, by means of a calculated GOTO, the same one as is used for moving forward.

Line 2000 to 2090 is the routine that handles drawing when you are facing east and is similar to the other three direction routines up to 2690. The loop m handles from your present position to the outer limit of the maze. X is what will be the co- ordinate across the screen of the bottom nearest to you of the wall which will next be drawn on the left of the screen. X will be used as a scaling factor as well for the perspective view. Lines 2025 to 2050 select whether to draw an open wall or a closed wall on the left or on the right. Line 2070 decides whether or not there is an obstruction ahead of you, in which case a wall is drawn ahead of you in line 3500. The short routines in 4000 to 7050 deal with the actual drawing of the walls on either side of you using 8 pixels wide as the width for each wall/opening (and the fact that each successive wall will be 12 pixels shorter as they get further away from you). This is the reason why the maze size is fixed at 15 by 15. Line 8000 is the start of the "help" routine which prints the plan of the maze then shows your position as an asterisk. Lines 8051 to 8054 determine where to mark the entrance and exit since it is possible to change these as described later. Line 8059 deducts the penalty points for demanding help. Line 8060 determines how much time you get to study the maze before the screen is cleared and the program continues by drawing your current positon in the maze (line 8070).

Line 8500 prints your score and direction (lines 8510 to 8515) in the lower screen using PRINT 1 to make it appear as though this writing is in the BORDER area out of the main display. Line 8505 ensures that the score never goes below zero. We come to line 9000 which is the main initialisation routine. Lines 9020 to 9180 must be entered with the line numbers as shown exactly if you wish to change the maze design later.

The DATA statements are laid out so that the maze design is obvious at a glance. ENTRYX is the co-ordinate across (1 to 15, left to right) of the entrance. ENTRYY is the Y co-ordinate (1 to 15, top to bottom) of the entrance. EXITX is the X co- ordinate of the exit from the maze and EXITY is the Y co- ordinate of this exit, as the variable names imply. Line 9190 sets up an array M$ (15,15) which holds the maze. This could be saved on tape using the SAVE"name"DATA facility but would, of course, be cleared if RUN was used, which is why the DATA statements will be used later for new mazes. Lines 9210 to 9230 read the maze from the DATA statements into the maze array then the start co-ordinates are set equal to the entrance co-ordinates so that you start in the entrance. D$ is set to N to indicate starting in a northerly direction in this case.

Line 9261 gives a short two note bleep to alert you that everything is ready to go. Line 9262 waits for you to let go of the keys if you are already pressing something. Lines 9263/4 wait for you to press any key (including SHIFTs). The score at the start is set up as 320 because the help routine is used for the initial preview of the maze and this will deduct 20 points making it 300 as we want. Lines 9266 to 9290 set up the display preview and score/direction, then draw the maze ahead in 3D.

Making changes to the program

The easiest change to make is the colours of the display, currently white on blue background with a black BORDER. Lines 10 and 20 are all that need to be changed.

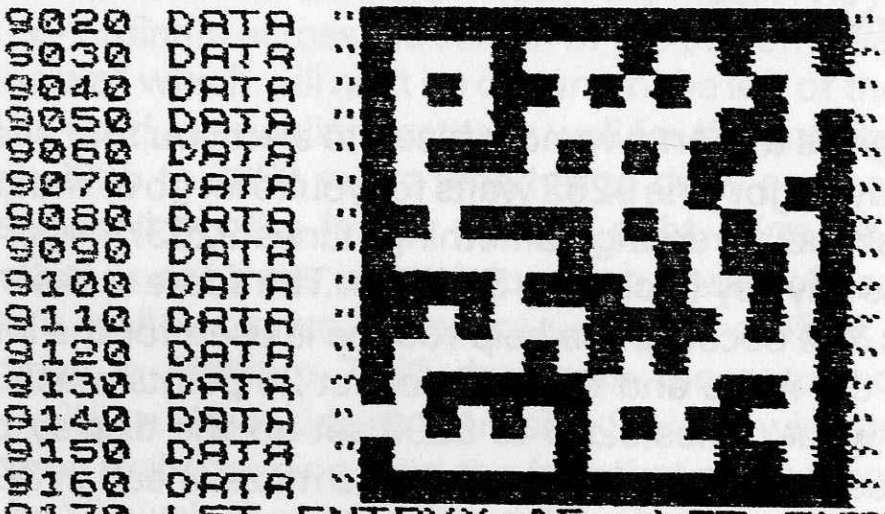
You can change the amount of time you get to view the display by changing the length of the PAUSE in line 8060. The character used to show your position can be any you like — it could

even be a space, but then you wouldn't get to see where you are although you would see the plan of the maze!

You can change the starting score in line 9265 if you're having a lot of trouble with getting out of the maze. And if you want to deduct more or less points for asking for help, change line 8059. Also, if you're more musically inclined than me you could arrange for a short tune to be played when you successfully get out of the maze. This would be done by replacing the STOP in line 140 with a GOTO to a routine to play the tune. This could be put between lines 200 and 1000 since this part is not used before the subroutines and after the main loop.

But the biggest change that can be made is to the maze. The best thing you can do is to save lines 9020 to 9180 separately on tape to be merged with the main program later on, this way you can quickly load in a new maze.

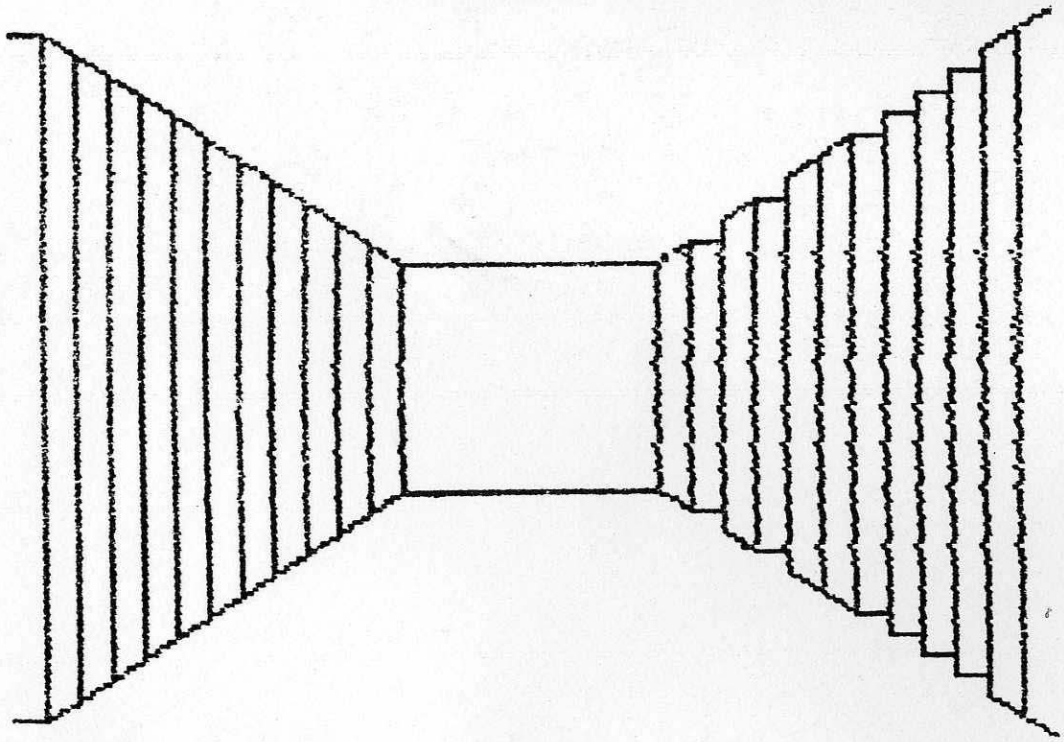
Here's an example:



```
9020 DATA "
9030 DATA "
9040 DATA "
9050 DATA "
9060 DATA "
9070 DATA "
9080 DATA "
9090 DATA "
9100 DATA "
9110 DATA "
9120 DATA "
9130 DATA "
9140 DATA "
9150 DATA "
9160 DATA "
9170 LET ENTRYX=15: LET ENTRYY=3
9175 LET D$="W"
9180 LET EXITX=1: LET EXITY=13
```

```
9020 DATA
9030 DATA
9040 DATA
9050 DATA
9060 DATA
9070 DATA
9080 DATA
9090 DATA
9100 DATA
9110 DATA
9120 DATA
9130 DATA
9140 DATA
9150 DATA
9160 DATA
9170 LET ENTRYX=1: LET ENTRYY=7
9175 LET D$="E"
9180 LET EXITX=1: LET EXITY=3
```

One thing to watch out for is not to have corridors more than one square wide. As you can see from this example where the first corridor to the right should be one corridor four times the width of a normal corridor, they are actually shown as four side by side corridors. Quite correct, but you may find it a little misleading at first.



A Tim Hartnell 'Success in the Fast Lane'* programming guide

When you've mastered introductory programming on the Spectrum, you need this outstanding guide to enhanced programming techniques and concepts.

Contents include:

- screen tricks
- escaping from INPUTs
- screen scrolling
- new character sets
- using the block graphics
- a library of subroutines
- user defined graphics
- sorting out SCREEN$ and ATTR
- IN and OUT
- speeding up your programs
- making use of the system variables
- memory layout
- useful DEF FN calls

Programs include INTRUDERS and 3D MAZE

Another great book from INTERFACE PUBLICATIONS London and Melbourne

AP 1400
F 3