A MARSHALL CAVENDISH 35 COMPUTER COURSE IN WEEKLY PARTS

INPUT

LEARN PROGRAMMING - FOR FUN AND THE FUTURE

UK £1.00
Republic of Ireland £1.25
Malta 85c
Australia $2.25
New Zealand $2.95

INPUT

Vol. 3 No 35

INDEX

The last part of INPUT, Part 52, will contain a complete, cross-referenced index. For easy access to your growing collection, a cumulative index to the contents of each issue is contained on the inside back cover.

PICTURE CREDITS
Front cover, Dave King. Pages 1081, 1083, 1084, 1086, Gary Wing. Pages 1082, 1092, 1100, Peter Reilly. Page 1088, Ian Craig. Page 1090, NASA. Pages 1091, David Redfern Photography, Paul Chave. Page 1094, Paul Chave. Pages 1096, 1098, 1100, Phil Dobson. Pages 1011, 1102, 1104, Steve Rippington. Pages 1107, 1111, Dave King.

Published by Marshall Cavendish Partworks Ltd, 58 Old Compton Street, London W1V 5PA, England. Typeset by MS Filmsetting Limited, Frome, Somerset. Printed by Cooper Clegg Web Offset Ltd, Gloucester and Howard Hunt Litho, London.

There are four binders each holding 13 issues.

HOW TO ORDER YOUR BINDERS

UK and Republic of Ireland:
Send £4.95 (inc p & p) (IR£5.95) for each binder to the address below:
Marshall Cavendish Services Ltd,
Department 980, Newtown Road,
Hove, Sussex BN3 7DN

Australia: See inserts for details, or write to INPUT, Times Consultants PO Box 213, Alexandria, NSW 2015

New Zealand: See inserts for details, or write to INPUT, Gordon and Gotch (NZ) Ltd, PO Box 1595, Wellington

Malta: Binders are available from local newsagents.

BACK NUMBERS

Back numbers are supplied at the regular cover price (subject to availability).

UK and Republic of Ireland:
INPUT, Dept AN, Marshall Cavendish Services,
Newtown Road, Hove BN3 7DN

Australia, New Zealand and Malta:
Back numbers are available through your local newsagent.

COPIES BY POST

Our Subscription Department can supply copies to any UK address regularly at £1.00 each. For example the cost of 26 issues is £26.00; for any other quantity simply multiply the number of issues required by £1.00. Send your order, with payment to:

Subscription Department, Marshall Cavendish Services Ltd,
Newtown Road, Hove, Sussex BN3 7DN

Please state the title of the publication and the part from which you wish to start.

HOW TO PAY: Readers in UK and Republic of Ireland: All cheques or postal orders for binders, back numbers and copies by post should be made payable to:
Marshall Cavendish Partworks Ltd.

QUERIES: When writing in, please give the make and model of your computer, as well as the Part No., page and line where the program is rejected or where it does not work. We can only answer specific queries – and please do not telephone. Send your queries to INPUT Queries, Marshall Cavendish Partworks Ltd, 58 Old Compton Street, London W1V 5PA.

INPUT IS SPECIALLY DESIGNED FOR:

The SINCLAIR ZX SPECTRUM (16K, 48K, 128 and +), COMMODORE 64 and 128, ACORN ELECTRON, BBC B and B+, and the DRAGON 32 and 64.

In addition, many of the programs and explanations are also suitable for the SINCLAIR ZX81, COMMODORE VIC 20, and TANDY COLOUR COMPUTER in 32K with extended BASIC. Programs and text which are specifically for particular machines are indicated by the following symbols:

SPECTRUM 16K, 48K, 128, and +
COMMODORE 64 and 128
ACORN ELECTRON, BBC B and B+
DRAGON 32 and 64
ZX81
VIC 20
TANDY TRS80 COLOUR COMPUTER

LOOKING INTO IT

- DRAWING A PICTURE
- SETTING THE SCALE
- MAGNIFYING AND REDUCING
- HIDING DETAIL IN THE PICTURE
- MAKING ACCURATE DRAWINGS

This versatile program can reduce a picture to a microscopic dot or expand it to astronomical dimensions. Use it to create highly accurate drawings or as a game

The program that accompanies this article is rather more fun than the usual applications program. In fact it can be used either as a game or an application.

It lets you draw a picture and then magnify any part of it to draw in more detail down to a microscopic scale. You can change the scale at any time using magnifications of many thousands or even hundreds of thousands. An enormous amount of detail can be hidden in a picture which disappears from view as you pan out, and reappears again as you zoom in.

Imagine yourself with a powerful microscope peering more and more closely at some part of a scene. Detail appears as the magnification increases and blurs into nothing as it decreases. Or you can look at it the other way. Starting with a view of your house you can imagine moving further and further away out into space. One by one your house, town, country and even earth itself disappears into a dot in the distance. Similar graphics effects can be achieved using the relatively short program given here.

One way of using the program is to turn it into a game for two people. One person draws a scene hiding some 'treasure' at a very small magnification somewhere deep in the picture. The other person then has to search for the treasure. It can be surprisingly difficult. Imagine starting with a square 10 centimetres across. If this is magnified 5000 times the original square turns into an enormous area of a quarter of a million square metres—plenty of room to hide almost anything in.

The program has serious applications too, of course. Very accurate line drawings can be made by entering detail at a much larger scale than usual and then reducing it to the correct size. Accurate technical drawings can be built up a section at a time in this way. The program is also useful in teaching. For example, in a geography lesson, the position of major towns could be marked on a map of the country but at a much reduced scale. At normal magnification they would only be a residual dot and would only appear in detail if you chose to zoom in at the correct position in the picture.

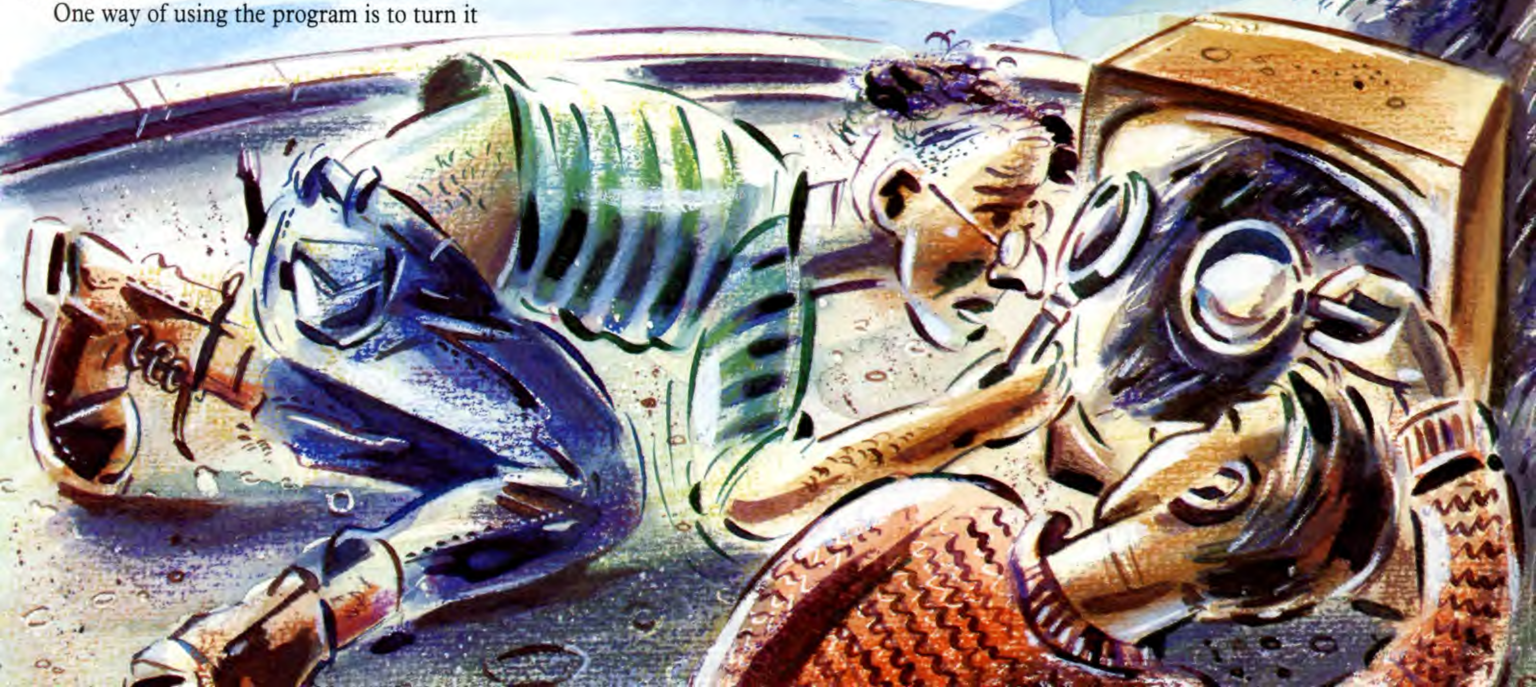
You could even enter the names of the towns—although you cannot enter text with this program, it is a simple enough matter to draw out the letters. Use a large scale to draw them first—in the correct position—then reduce them to the right size.

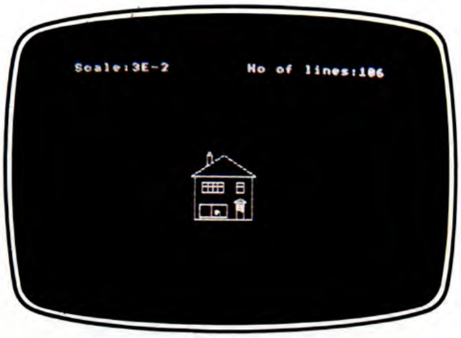
HOW TO USE THE PROGRAM

When you RUN the program you are presented with a blank screen and a tiny flashing cursor at the centre. There are also two numbers. The one on the left gives the scale of the present screen, 1 to start with, and the one on the right gives the number of lines drawn. This is dimensioned to a maximum of 600 at the start of the program. The program is controlled from the keyboard and is extremely easy to use.

On all but the Commodore, the cursor keys move the dot around the screen. The Commodore uses @ and / for up and down, and : and ; for left and right. You can speed up the movement of the cursor by simultaneously pressing SYMBOL SHIFT on the Spectrum, SHIFT on the Commodore, COPY on the Acorn and CLEAR on the Dragon and Tandy.

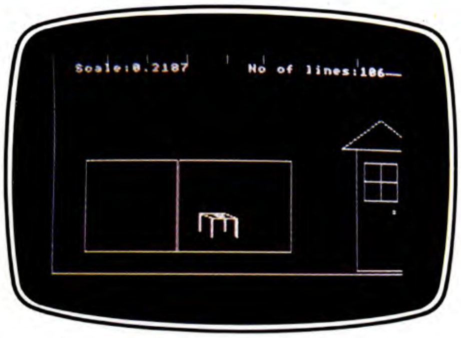
The other keys used are L, M, C, D, I, O and Z. Here's what they do. Press L to draw a line to the cursor or M to move to the cursor without drawing a line. Pressing C returns the cursor to the centre of the screen.





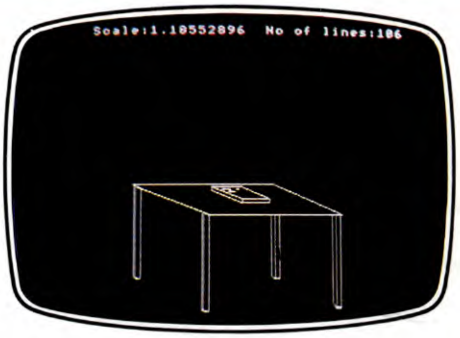
Scale:3E-2 No of lines:106

If you magnify this house



Scale:0.2187 No of lines:106

you can look through the window



Scale:1.10552896 No of lines:106

and see a book on the table

If you find you've made a mistake, press D for delete and you'll be asked how many lines you want to delete. The program counts backward from the last line drawn and deletes the correct number of lines. The O key outputs a file so you can save your drawing, and I inputs a file to load in a saved picture. The Acorn program works with both tape and disk. For the other computers, changes for disk are given after the main program.

Now for the interesting part. The Z (Zoom) key allows you to change the scale of the picture. The scale, or magnification, is entered as a number; for example, 2 doubles the scale and .5 halves it. The picture is redrawn to the new scale *centred on the cursor*—so always move the cursor to the middle of the point you want magnified. If any part of a line goes off the screen then it is not drawn at all except on the Acorns. The Acorns draw the part of the line that crosses the screen.

Zooming to a scale of Ø does nothing, while a scale of one redraws the picture centred on the cursor, but to the same scale. Entering a negative number produces a mirror image of the picture. The changes of scale are cumulative so a magnification of 2 followed by another 2 increases the magnification to 4.

Note that if you enter a very small scale on the Dragon and Tandy, say .ØØØ1, this is converted to 1ØE-4 but appears on the screen as 1Ø4. This is a limitation of the routine that draws the numbers on the screen and doesn't affect the way the picture is drawn.

```
10 DEF FN a(x)=(x/256)+128
20 DEF FN b(x)=(x/256)+85
30 BORDER Ø: INK 7: PAPER Ø: CLS
5Ø LET ln=1: LET sc=1: LET l=6ØØ
6Ø LET f=Ø: LET x=Ø: LET y=Ø: LET
   zoom=1
7Ø DIM a(l+1): DIM b(l): DIM c(l): DIM d(l)
8Ø LET a(1)=x: LET b(1)=y
9Ø LET c(1)=x: LET d(1)=y
1ØØ LET i$=INKEY$
11Ø IF i$="l" AND f AND l>ln THEN BEEP
   Ø.1,Ø: PLOT FN a(a(ln)),FN b(b(ln)): DRAW
   FN a(c(ln))-PEEK 23677,FN
   b(d(ln))-PEEK 23678: LET ln=ln+1: LET
   a(ln)=x: LET b(ln)=y: LET c(ln)=x: LET
   d(ln)=y: LET f=Ø
12Ø IF i$="c" THEN LET x=Ø: LET y=Ø:
   LET c(ln)=x: LET d(ln)=y
13Ø IF i$="m" THEN LET a(ln)=x: LET
   b(ln)=y
14Ø IF i$="i" THEN GOSUB 42Ø
15Ø IF i$="o" THEN GOSUB 6ØØ
16Ø IF i$="d" THEN GOSUB 71Ø
17Ø IF i$="z" THEN LET lv=Ø: GOSUB 84Ø
18Ø LET sp=256: IF CODE INKEY$<=41
   THEN LET sp=2Ø48
19Ø IF INKEY$="8" OR INKEY$="(" AND
   x>-32768+sp THEN LET f=1: LET
   x=x+sp: LET c(ln)=c(ln)+sp
2ØØ IF INKEY$="5" OR INKEY$="%" AND
   x<32768-sp THEN LET f=-1: LET
   x=x-sp: LET c(ln)=c(ln)-sp
21Ø IF INKEY$="7" OR INKEY$="'" AND
   y>-224ØØ+sp THEN LET f=-1: LET
   y=y+sp: LET d(ln)=d(ln)+sp
22Ø IF INKEY$="6" OR INKEY$="&" AND
   y<224ØØ-sp THEN LET f=1: LET
   y=y-sp: LET d(ln)=d(ln)-sp
23Ø GOSUB 25Ø
24Ø GOTO 1ØØ
25Ø PRINT AT 1,2; INVERSE 1;
   "□SCALE :";: LET a$=STR$ sc: IF LEN
   a$>4 THEN LET a$=a$( TO 4)
255 PRINT INVERSE 1;a$;AT 1,18;
   "□LINES:";ln-1
28Ø LET bx=FN a(a(ln)): LET by=FN
   b(b(ln)): LET ex=FN a(c(ln)): LET ey=FN
   b(d(ln))
29Ø PLOT bx,by: PLOT ex,ey
31Ø PLOT OVER 1;bx,by: PLOT OVER 1;ex,ey
32Ø RETURN
42Ø CLS
43Ø INPUT "ENTER FILENAME□";f$
435 IF f$="" THEN GOTO 43Ø
44Ø LET ln=1: LET zoom=1: LET sc=1
46Ø LOAD f$ DATA a()
47Ø LOAD f$ DATA b()
48Ø LOAD f$ DATA c()
485 LOAD f$ DATA d()
49Ø LET ln=a(6Ø1)
59Ø RETURN
6ØØ CLS
61Ø INPUT "ENTER FILENAME□";f$
615 IF f$="" THEN GOTO 61Ø
62Ø LET lv=1: GOSUB 84Ø
625 LET a(6Ø1)=ln
63Ø SAVE f$ DATA a()
64Ø SAVE f$ DATA b()
65Ø SAVE f$ DATA c()
66Ø SAVE f$ DATA d()
7ØØ RETURN
71Ø INPUT "ENTER NUMBER OF LINES TO
   DELETE"'k
75Ø IF k=Ø OR ln-k<=Ø THEN GOTO
   83Ø
755 LET ln=ln-k
76Ø LET x=a(ln): LET y=b(ln)
78Ø CLS
79Ø LET c(ln)=x: LET d(ln)=y
8ØØ IF ln=1 THEN GOTO 83Ø
81Ø IF ABS (c(ln-1))<32767 AND ABS
   (d(ln-1))<22399 THEN LET x=Ø: LET
   y=Ø
82Ø LET lv=2: GOSUB 84Ø
83Ø RETURN
84Ø IF ln=Ø THEN RETURN
85Ø IF lv=1 THEN LET zoom=1/sc: CLS :
   GOTO 92Ø
86Ø IF lv=2 THEN LET zoom=1: GOTO 92Ø
87Ø BEEP .1,1Ø: INPUT "ENTER ZOOM SCALE
   ";zoom
89Ø IF zoom=Ø THEN GOTO 1Ø2Ø
91Ø CLS
92Ø FOR u=1 TO ln-1
93Ø LET a(u)=(a(u)-x)*zoom: LET
   b(u)=(b(u)-y)*zoom
94Ø LET c(u)=(c(u)-x)*zoom: LET
   d(u)=(d(u)-y)*zoom
95Ø IF ABS (a(u))<32768 AND ABS
   (b(u))<224ØØ AND ABS (c(u))<32768
   AND ABS (d(u))<224ØØ THEN PLOT FN
```

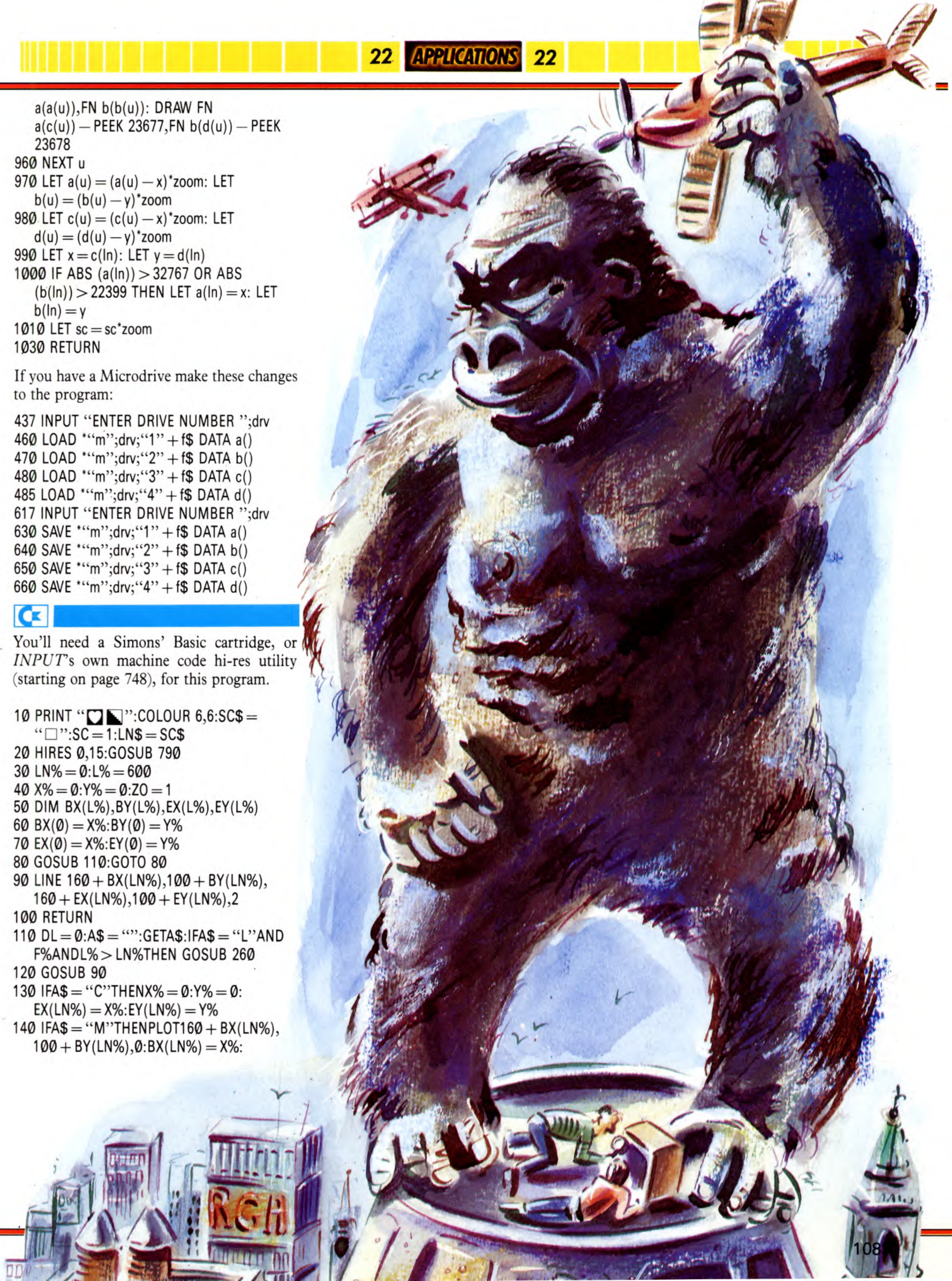
```
    a(a(u)),FN b(b(u)): DRAW FN
    a(c(u)) – PEEK 23677,FN b(d(u)) – PEEK
    23678
960 NEXT u
970 LET a(u) = (a(u) – x)*zoom: LET
    b(u) = (b(u) – y)*zoom
980 LET c(u) = (c(u) – x)*zoom: LET
    d(u) = (d(u) – y)*zoom
990 LET x = c(ln): LET y = d(ln)
1000 IF ABS (a(ln)) > 32767 OR ABS
    (b(ln)) > 22399 THEN LET a(ln) = x: LET
    b(ln) = y
1010 LET sc = sc*zoom
1030 RETURN
```

If you have a Microdrive make these changes to the program:

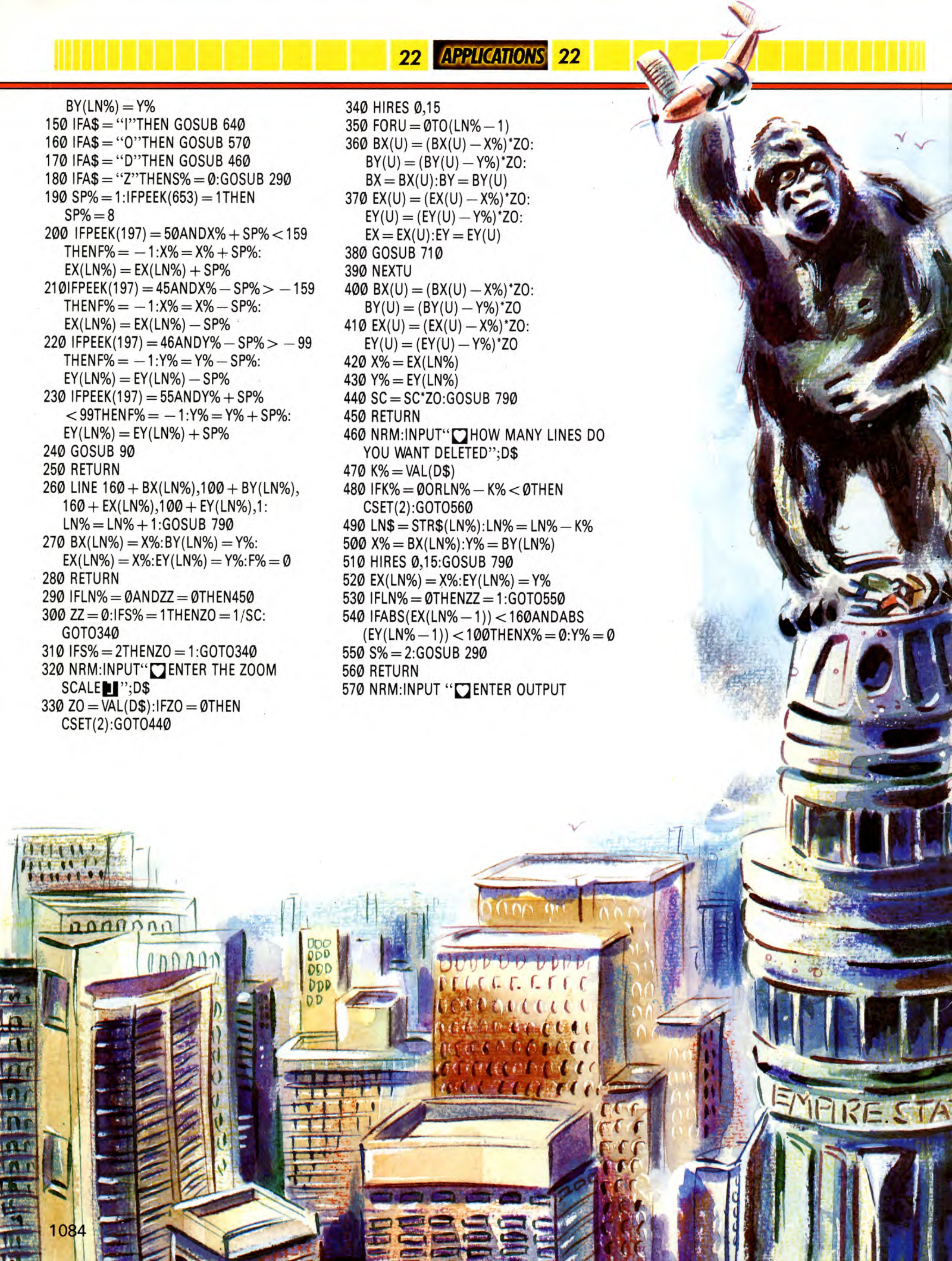
```
437 INPUT "ENTER DRIVE NUMBER ";drv
460 LOAD *"m";drv;"1" + f$ DATA a()
470 LOAD *"m";drv;"2" + f$ DATA b()
480 LOAD *"m";drv;"3" + f$ DATA c()
485 LOAD *"m";drv;"4" + f$ DATA d()
617 INPUT "ENTER DRIVE NUMBER ";drv
630 SAVE *"m";drv;"1" + f$ DATA a()
640 SAVE *"m";drv;"2" + f$ DATA b()
650 SAVE *"m";drv;"3" + f$ DATA c()
660 SAVE *"m";drv;"4" + f$ DATA d()
```

You'll need a Simons' Basic cartridge, or *INPUT*'s own machine code hi-res utility (starting on page 748), for this program.

```
10 PRINT "♥◣":COLOUR 6,6:SC$ =
    "□":SC = 1:LN$ = SC$
20 HIRES Ø,15:GOSUB 79Ø
30 LN% = Ø:L% = 6ØØ
40 X% = Ø:Y% = Ø:ZO = 1
50 DIM BX(L%),BY(L%),EX(L%),EY(L%)
60 BX(Ø) = X%:BY(Ø) = Y%
70 EX(Ø) = X%:EY(Ø) = Y%
80 GOSUB 11Ø:GOTO 8Ø
90 LINE 16Ø + BX(LN%),1ØØ + BY(LN%),
    16Ø + EX(LN%),1ØØ + EY(LN%),2
100 RETURN
110 DL = Ø:A$ = "":GETA$:IFA$ = "L"AND
    F%ANDL% > LN%THEN GOSUB 26Ø
120 GOSUB 9Ø
130 IFA$ = "C"THENX% = Ø:Y% = Ø:
    EX(LN%) = X%:EY(LN%) = Y%
140 IFA$ = "M"THENPLOT16Ø + BX(LN%),
    1ØØ + BY(LN%),Ø:BX(LN%) = X%:
```



```
    BY(LN%) = Y%
150 IFA$ = "I"THEN GOSUB 64Ø
160 IFA$ = "O"THEN GOSUB 57Ø
170 IFA$ = "D"THEN GOSUB 46Ø
180 IFA$ = "Z"THENS% = Ø:GOSUB 29Ø
190 SP% = 1:IFPEEK(653) = 1THEN
    SP% = 8
200 IFPEEK(197) = 5ØANDX% + SP% < 159
    THENF% = −1:X% = X% + SP%:
    EX(LN%) = EX(LN%) + SP%
21ØIFPEEK(197) = 45ANDX% − SP% > −159
    THENF% = −1:X% = X% − SP%:
    EX(LN%) = EX(LN%) − SP%
220 IFPEEK(197) = 46ANDY% − SP% > −99
    THENF% = −1:Y% = Y% − SP%:
    EY(LN%) = EY(LN%) − SP%
230 IFPEEK(197) = 55ANDY% + SP%
    < 99THENF% = −1:Y% = Y% + SP%:
    EY(LN%) = EY(LN%) + SP%
240 GOSUB 9Ø
250 RETURN
260 LINE 16Ø + BX(LN%),1ØØ + BY(LN%),
    16Ø + EX(LN%),1ØØ + EY(LN%),1:
    LN% = LN% + 1:GOSUB 79Ø
270 BX(LN%) = X%:BY(LN%) = Y%:
    EX(LN%) = X%:EY(LN%) = Y%:F% = Ø
280 RETURN
290 IFLN% = ØANDZZ = ØTHEN45Ø
300 ZZ = Ø:IFS% = 1THENZO = 1/SC:
    GOTO34Ø
310 IFS% = 2THENZO = 1:GOTO34Ø
320 NRM:INPUT"♡ENTER THE ZOOM
    SCALE▟";D$
330 ZO = VAL(D$):IFZO = ØTHEN
    CSET(2):GOTO44Ø
340 HIRES Ø,15
350 FORU = ØTO(LN% − 1)
360 BX(U) = (BX(U) − X%)*ZO:
    BY(U) = (BY(U) − Y%)*ZO:
    BX = BX(U):BY = BY(U)
370 EX(U) = (EX(U) − X%)*ZO:
    EY(U) = (EY(U) − Y%)*ZO:
    EX = EX(U):EY = EY(U)
380 GOSUB 71Ø
390 NEXTU
400 BX(U) = (BX(U) − X%)*ZO:
    BY(U) = (BY(U) − Y%)*ZO
410 EX(U) = (EX(U) − X%)*ZO:
    EY(U) = (EY(U) − Y%)*ZO
420 X% = EX(LN%)
430 Y% = EY(LN%)
440 SC = SC*ZO:GOSUB 79Ø
450 RETURN
460 NRM:INPUT"♡HOW MANY LINES DO
    YOU WANT DELETED";D$
470 K% = VAL(D$)
480 IFK% = ØORLN% − K% < ØTHEN
    CSET(2):GOTO56Ø
490 LN$ = STR$(LN%):LN% = LN% − K%
500 X% = BX(LN%):Y% = BY(LN%)
510 HIRES Ø,15:GOSUB 79Ø
520 EX(LN%) = X%:EY(LN%) = Y%
530 IFLN% = ØTHENZZ = 1:GOTO55Ø
540 IFABS(EX(LN% − 1)) < 16ØANDABS
    (EY(LN% − 1)) < 1ØØTHENX% = Ø:Y% = Ø
550 S% = 2:GOSUB 29Ø
560 RETURN
570 NRM:INPUT "♡ENTER OUTPUT
```



```
   FILENAME▮";F$
580 S%=1:GOSUB 290
590 NRM:CM$=",":OPEN1,1,1,F$
600 PRINT#1,LN%:FORU=ØTOLN%
610 PRINT#1,BX(U);CM$;BY(U);CM$;EX(U);
   CM$;EY(U)
620 NEXTU:CLOSE1:HIRES Ø,15:
   S%=2:GOSUB 290
630 RETURN
640 NRM:INPUT"♥ENTER INPUT
   FILENAME▮";F$
650 LN%=Ø:S%=1:GOSUB 290
660 NRM:OPEN1,1,Ø,F$
670 INPUT#1,LN%:FORU=ØTOLN%
680 INPUT#1,BX(U),BY(U),EX(U),EY(U):
   NEXTU
690 CLOSE1:X%=EX(LN%):Y%=EY(LN%):
   HIRES Ø,15:GOSUB 79Ø:S%=2:
   GOSUB 290
700 RETURN
710 SX=(EX-BX)/100:SY=
   (EY-BY)/100
720 FORQ=ØTO100
730 IFABS(BX)>155ORABS(BY)>95THEN
   BX=BX+SX:BY=BY+SY:NEXTQ:
   GOTO780
740 SX=(EX-BX)/100:SY=(EY-BY)/100
750 FORQ=ØTO100
760 IFABS(EX)>155ORABS(EY)>95THEN
   EX=EX-SX:EY=EY-SY:NEXTQ
770 LINE 160+BX,100+BY,160+EX,
   100+EY,1
780 RETURN
790 TEXT 264,Ø,LN$,Ø,1,8:LN$=STR$(LN%):
   TEXT 16Ø,Ø,"NUM OF LINES:
   "+LN$,1,1,8
800 TEXT 48,Ø,SC$,Ø,1,8:SC$=STR$(INT
   (SC*1000)/1000):
   TEXT Ø,Ø,"SCALE:"+SC$,1,1,8
810 RETURN
```

To use the program with a disk drive change Lines 59Ø and 66Ø to:

```
590 NRM: CM$=",":OPEN1,8,1,F$
660 NRM:OPEN1,8,Ø,F$
```

```
10 *FX212,216
20 ON ERROR GOTO 1260
30 MODE4:VDU 23,1,Ø;Ø;Ø;Ø;:
   VDU29,640;512;
40 L%=Ø:scale=1:N%=600
50 X%=Ø:Y%=Ø:zoom=1
60 DIM begx(N%),begy(N%),
   endx(N%),endy(N%)
70 begx(Ø)=X%:begy(Ø)=Y%
80 endx(Ø)=X%:endy(Ø)=Y%
90 *FX4,1
95 PLOT 69,Ø,Ø
100 REPEAT
110 PROCkey
120 UNTIL FALSE
130 END
140 DEF PROCdraw
150 PRINTTAB(5,1)"Scale:";scale;
   TAB(18);"No of lines:";L%
160 GCOL 4,Ø
170 MOVE begx(L%),begy(L%):
   DRAW endx(L%),endy(L%)
```

Microtip

Game tips

When you use this program as a game, the difficult part is to make sure that the hidden message or treasure doesn't appear as an obvious blob in the picture.

The trick is to draw in lots of detail at different magnifications, so your opponent has to search for the correct area of the picture as well as the correct magnification. Don't hide the treasure too well, though, as it is easy to become hopelessly lost at higher magnifications.

If you want to score the game, simply count the number of moves and changes of scale the player makes. The one with the smallest number is the winner.

```
175 PLOT 69,begx(L%),begy(L%)
18Ø ENDPROC
19Ø DEF PROCkey
2ØØ IF INKEY(-87)=-1 AND F% AND
   N%>L% VDU7:GCOLØ,1:MOVE begx(L%),
   begy(L%):DRAW endx(L%),endy(L%):L%=
   L%+1:begx(L%)=X%:begy(L%)=Y%:
   endx(L%)=X%:endy(L%)=Y%:F%=Ø
21Ø PROCdraw
22Ø IF INKEY(-83) VDU7:
   X%=Ø:Y%=Ø:endx(L%)=X%:
   endy(L%)=Y%
23Ø IF INKEY(-1Ø2) VDU7:
   begx(L%)=X%:begy(L%)=Y%
24Ø IF INKEY(-38) PROCinput
25Ø IF INKEY(-55) PROCoutput
26Ø IF INKEY(-51) PROCdelete
27Ø IF INKEY(-98) PROCzoom(Ø)
28Ø IF INKEY(-1Ø6) sp%=24 ELSE sp%=4
29Ø IF INKEY(-122) F%=-1:
   X%=X%+sp%:endx(L%)=
   endx(L%)+sp%
3ØØ IF INKEY(-26) F%=-1:
   X%=X%-sp%:endx(L%)=
   endx(L%)-sp%
31Ø IF INKEY(-42) F%=-1:
   Y%=Y%-sp%:endy(L%)=
   endy(L%)-sp%
32Ø IF INKEY(-58) F%=-1:
   Y%=Y%+sp%:endy(L%)=
   endy(L%)+sp%
33Ø PROCdraw
34Ø ENDPROC
35Ø DEF PROCzoom(s%)
36Ø IF L%=Ø GOTO 66Ø
37Ø IF s%=1 THEN zoom=1/scale:
   CLS:GOTO 54Ø
38Ø IF s%=2 THEN zoom=1:GOTO 54Ø
39Ø VDU7
4ØØ VDU3Ø:PRINT'SPC(39):VDU3Ø
41Ø PRINT'"Enter the zoom scale□";
42Ø D$=""
43Ø REPEAT
44Ø FOR C%=Ø TO1ØØ:NEXT:*FX15,1
45Ø zoom=GET
46Ø IF zoom=127 AND LEN(D$)=Ø
   VDU7:GOTO 44Ø
47Ø IF zoom=127 D$=LEFT$
   (D$,LEN(D$)-1):GOTO 5ØØ
48Ø IF LEN(D$)=19 VDU7:GOTO 44Ø
485 IF (zoom<46 OR zoom>57) AND
   zoom<>13 AND zoom<>69
   THEN 45Ø
49Ø D$=D$+CHR$(zoom)
5ØØ VDU zoom
51Ø UNTIL zoom=13
52Ø zoom=EVAL(D$)
53Ø IF zoom=Ø VDU3Ø:PRINT'
   SPC(39):GOTO 65Ø
54Ø CLS:GCOLØ,1
55Ø FOR U%=Ø TO L%-1
56Ø begx(U%)=(begx(U%)-X%)*zoom:begy
   (U%)=(begy(U%)-Y%)*zoom
57Ø endx(U%)=(endx(U%)-X%)*zoom:
   endy(U%)=(endy(U%)-Y%)*zoom
58Ø IF ABS(begx(U%))<32768 AND ABS
   (begy(U%))<32768 AND ABS(endx(U%))
   <32768 AND ABS(endx(U%))<32768
   MOVE begx(U%),begy(U%):DRAW endx
   (U%),endy(U%)
59Ø NEXT
6ØØ begx(U%)=(begx(U%)-X%)*zoom:begy
   (U%)=(begy(U%)-Y%)*zoom
61Ø endx(U%)=(endx(U%)-X%)*zoom:endy
   (U%)=(endy(U%)-Y%)*zoom
62Ø X%=endx(L%):Y%=endy(L%)
63Ø IF ABS(begx(L%))>32767 OR
   ABS(begy(L%))>32767 THEN
   begx(L%)=X%:begy(L%)=Y%
64Ø scale=scale*zoom
65Ø VDU7
66Ø ENDPROC
67Ø DEF PROCdelete
68Ø VDU7
69Ø VDU3Ø:PRINT'SPC(39):VDU3Ø
7ØØ PRINT'"How many lines do you want
   deleted□";
71Ø D$=""
72Ø REPEAT
73Ø FOR C%=Ø TO1ØØ:NEXT:*FX15,1
74Ø K%=GET
75Ø IF K%=13 GOTO 81Ø
76Ø IF K%<48 OR K%>57 AND
   (K%<>127 OR LEN(D$)=Ø)
   VDU7:GOTO 73Ø
77Ø IF K%=127 D$=LEFT$(D$,
   LEN(D$)-1):GOTO 8ØØ
78Ø IF LEN(D$)=3 VDU7:GOTO 73Ø
79Ø D$=D$+CHR$(K%)
8ØØ VDU K%
81Ø UNTIL K%=13
82Ø K%=VAL(D$)
83Ø IF K%=Ø OR L%-K%<Ø GOTO
   91Ø
84Ø L%=L%-K%
85Ø X%=begx(L%):Y%=begy(L%)
86Ø CLS
87Ø endx(L%)=X%:endy(L%)=Y%
88Ø IF L%=Ø GOTO 91Ø
89Ø IF ABS(endx(L%-1)) <64Ø AND
   ABS(endy(L%-1)) <512 THEN
   X%=Ø:Y%=Ø
9ØØ PROCzoom(2)
91Ø VDU 3Ø:PRINT'SPC(39)
92Ø ENDPROC
93Ø DEF PROCinput
94Ø VDU3Ø:PRINT'SPC(39):VDU3Ø
95Ø VDU7:*FX15,1
96Ø INPUT'"Enter the input
   filename□"f$:CLS
97Ø L%=Ø:GCOLØ,Ø
98Ø PROCzoom(1)
99Ø GCOL4,Ø
1ØØØ D%=OPENUP f$
1Ø1Ø INPUT #D%,L%
1Ø2Ø FOR U%=Ø TO L%-1
1Ø3Ø INPUT #D%,begx(U%),begy(U%),
    endx(U%),endy(U%)
1Ø4Ø IF ABS(begx(U%))<32768 AND
    ABS(begy(U%))<32768 AND
    ABS(endx(U%))<32768 AND
    ABS(endy(U%))<32768 MOVE
    begx(U%),begy(U%): DRAW
    endx(U%),endy(U%)
1Ø5Ø NEXT
1Ø6Ø INPUT #D%,begx(L%),begy
    (L%),endx(L%),endy(L%)
1Ø7Ø X%=endx(L%):Y%=endy(L%)
1Ø8Ø CLOSE#Ø
1Ø9Ø VDU7
11ØØ ENDPROC
111Ø DEF PROCoutput
112Ø VDU3Ø:PRINT'SPC(39):VDU3Ø
113Ø VDU7:*FX15,1
114Ø INPUT'"Enter the output
    filename□"f$
115Ø PROCzoom(1)
116Ø VDU3Ø:PRINT'SPC(39):
    PRINTTAB(Ø,1);
117Ø D%=OPENOUT f$
118Ø PRINT #D%,L%
119Ø FOR U%=Ø TO L%
12ØØ PRINT #D%,begx(U%),begy
    (U%),endx(U%),endy(U%)
121Ø NEXT
122Ø CLOSE#Ø
123Ø VDU3Ø:PRINT'SPC(39)
124Ø VDU7
125Ø ENDPROC
126Ø MODE7
```

```
1270 *FX4,Ø
1280 REPORT:PRINT“□at line□”;ERL
1290 END
```

This program will RUN on the Tandy if you change the 223 in Line 1Ø to 247.

```
10 PMODE4,1:COLOR Ø,1:PCLS:
    SCREEN1,Ø:V=223
20 DIM NU$(1Ø):FOR I=Ø TO 1Ø:
    READ NU$(I):NEXT
30 DEF FNA(X)=(X/256)+128
40 DEF FNB(X)=(X/256)+96
50 LN=Ø:SC=1:L=6ØØ
60 X=Ø:Y=Ø:ZOOM=1
70 DIM BX(L),BY(L),EX(L),EY(L)
80 BX(Ø)=X:BY(Ø)=Y
90 EX(Ø)=X:EY(Ø)=Y
100 I$=INKEY$
110 IF I$=“L” AND F AND L>LN THEN
    LINE(FNA(BX(LN)),FNB(BY(LN)))-(FNA
    (EX(LN)),FNB(EY(LN))),
    PSET:LN=LN+1:BX(LN)=X:BY(LN)=Y:
    EX(LN)=X:EY(LN)=Y:F=Ø
120 IF I$=“C” THEN X=Ø:Y=Ø:
    EX(LN)=X:EY(LN)=Y
130 IF I$=“M” THEN BX(LN)=X:
    BY(LN)=Y
140 IF I$=“I” GOSUB 42Ø
150 IF I$=“O” GOSUB 6ØØ
160 IF I$=“D” GOSUB 71Ø
170 IF I$=“Z” THEN LV=Ø:GOSUB 84Ø
180 IF PEEK(339)=191 THEN SP=2Ø48
    ELSE SP=256
190 IF PEEK(343)=V AND X>-32768+SP
    THEN F=-1:X=X-SP:EX(LN)=
    EX(LN)-SP
200 IF PEEK(344)=V AND X<32768-SP
    THEN F=-1:X=X+SP:EX(LN)=
    EX(LN)+SP
210 IF PEEK(341)=V AND Y>-24576
    +SP THEN F=-1:Y=Y-SP:
    EY(LN)=EY(LN)-SP
220 IF PEEK(342)=V AND Y<24576-SP
    THEN F=-1:Y=Y+SP:EY(LN)=
    EY(LN)+SP
230 GOSUB 25Ø
240 GOTO 1ØØ
250 SS$=STR$(SC)
260 DRAW “BMØ,183”:GOSUB 33Ø
270 SS$=STR$(LN):DRAW “BM2ØØ,
    183”:GOSUB 33Ø
280 BX=FNA(BX(LN)):BY=FNB(BY(LN)):
    EX=FNA(EX(LN)):EY=FNB(EY(LN))
290 PSET(BX,BY):PSET(EX,EY)
300 FOR D=1 TO 5Ø:NEXTD
310 PRESET(BX,BY):PRESET(EX,EY)
320 RETURN
330 FOR I=1 TO LEN(SS$)
340 DI=ASC(MID$(SS$,I,1))-48
350 IF DI=-2 THEN DI=1Ø
360 IF DI<Ø OR DI>1Ø THEN 38Ø
370 DRAW “C1;XNU$(8);CØ;BL8”+
    NU$(DI)+“BR2”
380 NEXT I
390 RETURN
400 DATA R6D8L6U8BR8,BR6ND8BR2,
    R6D4L6D4R6BR2BU8,
    R6D4NL3D4NL6BR2BU8,D4R6D4U8BR2,
    NR6D4R6D4L6BE8
410 DATA D8R6U4L6U4BR8,R6ND8BR2,
    R6D8L6U8D4R6U4BR2,D4R6D4U8L6BR8,
    BR3BD8NR1BR5BU8
420 CLS
430 PRINT@256,“”;:LINE INPUT “ENTER
    LOAD FILENAME>□”;F$
440 LN=Ø:ZOOM=1:SC=1
450 PCLS:SCREEN1,Ø
460 OPEN “I”,#-1,F$
470 INPUT #-1,N$
480 LN=VAL(N$)
490 FOR U=Ø TO LN-1
500 INPUT #-1,BX$,BY$,EX$,EY$
510 BX(U)=VAL(BX$):BY(U)=VAL(BY$):
    EX(U)=VAL(EX$):EY(U)=VAL(EY$)
520 IF ABS(BX(U))<32768 AND
    ABS(BY(U))<24576 AND
    ABS(EX(U))<32768 AND
    ABS(EY(U))<24576 THEN LINE
    (FNA(BX(U)), FNB(BY(U)))-
    (FNA(EX(U)),FNB(EY(U))),
    PSET
530 NEXT
540 INPUT #-1,BX$,BY$,EX$,EY$
550 BX(U)=VAL(BX$):BY(U)=VAL
    (BY$):EX(U)=VAL(EX$):
    EY(U)=VAL(EY$)
560 X=EX(LN):Y=EY(LN)
570 CLOSE#-1
580 SCREEN1,Ø
590 RETURN
600 CLS
610 PRINT@256,“”;:LINE INPUT “ENTER
    SAVE FILENAME>□”;F$
620 LV=1:GOSUB 84Ø
630 OPEN “O”,#-1,F$
640 PRINT#-1,STR$(LN)
650 FOR U=Ø TO LN
660 PRINT#-1,STR$(BX(U)),
    STR$(BY(U)),STR$(EX(U)),
    STR$(EY(U))
670 NEXT
680 CLOSE#-1
690 SCREEN1,Ø
700 RETURN
710 CLS
720 PRINT@256,“HOW MANY LINES DO
    YOU WANT TO□□□DELETE ”;
730 INPUT K
740 SCREEN1,Ø
750 IF K=Ø OR LN-K<Ø THEN 83Ø
760 LN=LN-K
770 X=BX(LN):Y=BY(LN)
780 PCLS
790 EX(LN)=X:EY(LN)=Y
800 IF LN=Ø THEN 83Ø
810 IF ABS(EX(LN-1))<32767 AND
    ABS(EY(LN-1))<24576 THEN
    X=Ø:Y=Ø
820 LV=2:GOSUB 84Ø
830 SCREEN1,Ø:RETURN
840 IF LN=Ø THEN RETURN
850 IF LV=1 THEN ZOOM=1/SC:
    PCLS:GOTO 92Ø
860 IF LV=2 THEN ZOOM=1:GOTO 92Ø
870 CLS
880 PRINT@256,“ENTER THE ZOOM
    SCALE□”;
890 INPUT ZOOM
900 IF ZOOM=Ø THEN 1Ø2Ø
910 PCLS:SCREEN1,Ø
920 FOR U=Ø TO LN-1
930 BX(U)=(BX(U)-X)*ZOOM:
    BY(U)=(BY(U)-Y)*ZOOM
940 EX(U)=(EX(U)-X)*ZOOM:
    EY(U)=(EY(U)-Y)*ZOOM
950 IF ABS(BX(U))<32768 AND ABS
    (BY(U))<24576 AND ABS(EX(U))
    <32768 AND ABS(EY(U))<24576 THEN
    LINE(FNA(BX(U)),
    FNB(BY(U)))-(FNA(EX(U)),
    FNB(EY(U))),PSET
960 NEXT
970 BX(U)=(BX(U)-X)*ZOOM:
    BY(U)=(BY(U)-Y)*ZOOM
980 EX(U)=(EX(U)-X)*ZOOM:
    EY(U)=(EY(U)-Y)*ZOOM
990 X=EX(LN):Y=EY(LN)
1000 IF ABS(BX(LN))>32767 OR ABS
    (BY(LN))>24576 THEN BX(LN)=X:
    BY(LN)=Y
1010 SC=SC*ZOOM
1020 SCREEN1,Ø
1030 RETURN
```

The Dragon program will work with a Dragon Data disk drive if you make these changes:

Delete Line 46Ø and add

```
470 FREAD F$,FROMØ;N$
500 FREADF$;BX$,BY$,EX$,EY$
540 FREADF$;BX$,BY$,EX$,EY$
570 CLOSE
630 CREATE F$
640 FWRITEF$;STR$(LN)
660 FWRITEF$;STR$(BX(U)),“:”,
    STR$(BY(U)),“:”,STR$(EX(U)),
    “:”,STR$(EY(U))
680 CLOSE
```

LUNAR TOUCHDOWN

One small step for *INPUT*. . . . You'll need all your skill and judgement to manoeuvre the Lunar Lander safely to the landing pad, in this complete arcade-type game

Games Programming need not involve huge complicated programs to produce a self-contained game. Here is a version of the popular Lunar Lander program that offers high resolution graphics and full control over the craft.

The game is complete and playable as it stands, but there are many possibilities for you to customize it to suit your own preferences. You may well want to add an 'another go?' routine to save having to RUN the program after each descent. You may wish to alter the graphics and the sound too—it's up to you.

CONTROLS

Spectrum: the cursor keys—5, 7 and 8.
Commodore: the cursor keys for left and right, and the Commodore key for thrust.
Acorn: Z for left, X for right and space for thrust.
Dragon/Tandy: arrow control keys.

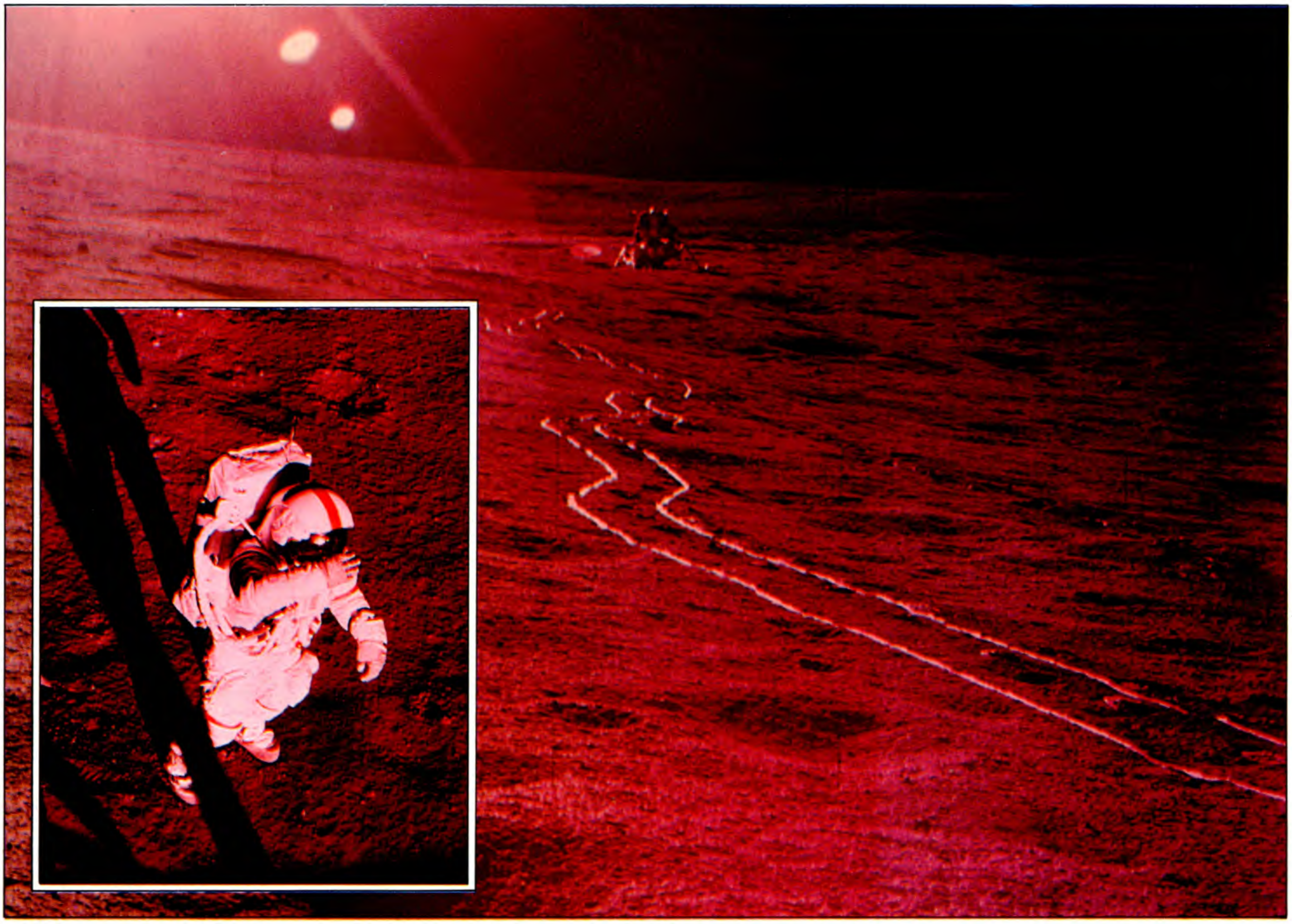
```
10 BORDER 1: INK 7: PAPER Ø: CLS : BRIGHT
   1
20 FOR N=1 TO 5Ø: PLOT RND*255,
   (RND*135)+4Ø: NEXT N
7Ø PLOT Ø,Ø: FOR N=1 TO 16: READ GX,GY:
   DRAW GX,GY: NEXT N
8Ø DATA 18,3Ø,18,−15,18,−8,18,8,16,2Ø,
   16,5,13,−2Ø,16,−8,18,−4,15,Ø,1Ø,1Ø,
   2Ø,25,1Ø,−2Ø,1Ø,−1Ø,2Ø,−5,18,2Ø
9Ø PRINT AT Ø,4; INK 6; PAPER
   2;" FUEL :";AT Ø,18;" VELOCITY :"
11Ø LET LX=RND*24Ø+1Ø: LET LY=
   16Ø−(15+(RND*1Ø)): LET XV=RND*15:
   LET YV=Ø: LET F=246
115 GOSUB 4ØØØ
12Ø GOSUB 1ØØØ: GOSUB 2ØØØ: GOSUB
   3ØØØ
13Ø IF LY>2Ø THEN GOTO 12Ø
135 PAUSE 5Ø
14Ø CLS : IF LX<154 OR LX>164 OR ABS
   YV>4 THEN GOTO 16Ø
15Ø PRINT "CONGRATULATIONS A
   SUCCESSFUL□□□□LANDING !!":
   RESTORE 5ØØØ: FOR N=1 TO 14: READ
   A,B: BEEP A,B: NEXT N: GOTO 17Ø
16Ø PRINT AT 1Ø,7; FLASH 1; INK 2; PAPER
   7;" !!!! CRASHED !!!! ": FOR T=1 TO 5Ø:
   BORDER RND*7: BEEP .Ø1,RND*5: NEXT T
17Ø PAUSE 4ØØ
18Ø STOP
1ØØØ IF LY<16Ø THEN GOSUB 4ØØØ
1Ø1Ø LET LX=LX+XV: LET LY=LY+YV: IF
   LY<3ØØ THEN BEEP .Ø2,LY/5
1Ø3Ø IF LX<5 THEN LET LX=LX+245
1Ø35 IF LX>25Ø THEN LET LX=LX−242
1Ø36 IF LY>16Ø THEN RETURN
1Ø37 IF LY<1Ø THEN RETURN
1Ø4Ø GOSUB 4ØØØ
1Ø5Ø RETURN
2ØØØ LET YV=YV−.5: IF F<1 THEN
   RETURN
2Ø1Ø IF INKEY$="7" AND F>3 THEN LET
   YV=YV+1: LET F=F−3: RETURN
2Ø2Ø IF INKEY$="5" THEN LET
   XV=XV−.5: LET F=F−1: RETURN
2Ø3Ø IF INKEY$="8" THEN LET
   XV=XV+.5: LET F=F−1: RETURN
2Ø4Ø RETURN
3ØØØ PRINT AT Ø,1Ø;"  "+STR$ F+"  ";AT
   Ø,28;"  "+STR$ INT YV+"  "
3Ø1Ø RETURN
4ØØØ OVER 1: PLOT LX,LY: DRAW −5,−1Ø:
   DRAW 5,2: DRAW 5,−2: DRAW −4,1Ø
4Ø1Ø OVER Ø: RETURN
5ØØØ DATA .2,4,.2,7,.2,5,.2,12,.2,Ø,.2,4,.2,4,
   .2,5,.6,7,.2,12,.2,Ø,.2,4,.2,2,.6,Ø
```

```
1Ø HIRES 1,Ø:MULTI 7,4,4:COLOUR 7,Ø:
   POKE 54296,15:POKE 54277,19Ø:
   POKE 54278,248
15 POKE 54276,Ø
7Ø FOR Z=2Ø TO 16Ø STEP 2Ø:Y=
   179−RND(1)*4Ø:LINE Z−2Ø,179,
   Z−1Ø,Y,3
72 FOR ZZ=1 TO 3:PLOT RND(1)*16Ø,
   RND(1)*15Ø,ZZ:NEXT ZZ
74 LINE Z−1Ø,Y,Z,179,3:NEXT Z:
   PAINT Ø,199,1:TEXT 64,192,
   "▨□□▧",2,1,8
76 LINE Ø,199,159,199,2
```

- ■ A COMPLETE GAME
- ■ LUNAR GRAPHICS
- ■ FUEL GAUGE
- ■ SPEED GAUGE
- ■ LANDING CONTROL
- ■ CUSTOMIZING THE PROGRAM
- ■ SOUND EFFECTS
- ■ CRASHES
- ■ SUCCESSFUL PROGRAMS
- ■ THE LANDER

```
110 LX=RND(1)*248:LY=15+RND
   (1)*10:XV=RND(1)*15-8:YV=Ø
   F=246
115 GOSUB 1Ø4Ø
12Ø GOSUB 1Ø1Ø:GOSUB 2ØØØ:MULTI
   7,RND(1)*2+4,RND(1)*8+1
13Ø IF LY<192 THEN 12Ø
14Ø IF LX<72 OR LX>88 OR YV>4 THEN
   16Ø
15Ø PRINT "♥▒ ▙▐▚
   CONGRATULATIONS A SUCCESSFUL
   LANDING !"
155 POKE 54276,33:FOR Z=1 TO 255:
   POKE 54273,Z:POKE54273,255-Z:
   NEXT Z:GOTO 17Ø
16Ø PRINT"♥▒BAD LANDING. . .":
   POKE 54276,129
163 FOR Z=1 TO 1ØØ:COLOUR 7,RND
   (1)*2:POKE 54273,Z
165 FOR ZZ=1 TO 1Ø:NEXT ZZ,Z
17Ø POKE 54276,Ø:NRM:POKE 198,Ø:END
1Ø1Ø LX=LX+XV:LY=LY+YV:POKE
   54276,17:S=255-(255ANDLY):
   POKE 54273,S
1Ø2Ø IF LY<13 THEN RETURN
1Ø3Ø IF LX<Ø THEN LX=151
1Ø35 IF LX>151 THEN LX=Ø
1Ø4Ø TEXT LX,LY,"♣",4,1,8:
   GOSUB 3ØØØ
1Ø5Ø TEXT LX,LY,"♣",4,1,8:
   POKE 54276,16:RETURN
2ØØØ YV=YV+.5:IF LY<13 THEN RETURN
2Ø1Ø GET K$:IF PEEK(653)=2 AND F>3
   THEN YV=YV-1:F=F-3:RETURN
2Ø2Ø IF K$="▙" THEN XV=XV-.5:
   F=F-1:RETURN
2Ø3Ø IF K$="▐" THEN XV=XV+.5:
   F=F-1
2Ø4Ø RETURN
3ØØØ TEXT 1,1,"FUEL:"+STR$(F),1,1,8
3Ø1Ø V=2*YV:IF ABS(V)>122 THEN
   V=122*SGN(V)
3Ø2Ø TEXT 75,1,"SPEED:"+STR$(V),1,1,8
3Ø21 TEXT 41,1,STR$(F),Ø,1,8
3Ø22 TEXT 123,1,STR$(V),Ø,1,8
3Ø3Ø RETURN
```

```
1Ø MODE1
2Ø GCOLØ,3
5Ø VDU 23,255,24,6Ø,9Ø,126,126,9Ø,129,129
7Ø MOVEØ,Ø:FOR X=8Ø TO 128Ø
   STEP8Ø:READ A:DRAWX,A*32:NEXT
8Ø DATA 6,2,1,3,8,9,4,2,1,Ø,Ø,2,9,
   4,1,7,3
9Ø PRINTTAB(Ø,1)"FUEL"'
   "SPEED"
11Ø LX=RND(1192):LY=RND
   (2ØØ)+69Ø:XV=RND(15)-8:YV=Ø:
   F=99Ø:MOVEX,LY:VDU5,255,4
115 MOVE2ØØ,96Ø:MOVE2ØØ,992:PLOT85,
   12ØØ,96Ø:PLOT85,12ØØ,992
116 GCOL3,3
12Ø GOSUB 1ØØØ:GOSUB2ØØØ:GOSUB3ØØØ
13Ø IF LY>32 THEN 12Ø
14Ø CLS:IF LX<8ØØ OR LX>848 OR
   YV< -8 THEN16Ø
15Ø PRINTTAB(Ø,7)"A SUCCESSFUL
   LANDING!!":GOTO 17Ø
16Ø PRINTTAB(Ø,7)"BAD CRASH"
17Ø FOR T=1 TO 4ØØØ:NEXT:*FX15,1
18Ø END
1ØØØ IF LY<9Ø1 THEN MOVELX,LY:VDU
   5,255,4
1Ø1Ø LX=LX+XV:LY=LY+YV:
   SOUND1,-15,LY/4,1
1Ø2Ø IF LY>9ØØ THEN RETURN
1Ø3Ø IF LX<Ø OR LX>1191 THEN
   LX=-1191*(LX<Ø)
1Ø4Ø MOVELX,LY:VDU5,255,4
2ØØØ YV=YV-.5:IF
F<1 THEN RETURN
2Ø1Ø IF INKEY(-99)
THEN YV=YV+1:
```



```
    F=F-3
2Ø2Ø IF INKEY(-98) THEN XV=XV-.5:
    F=F-1
2Ø3Ø IF INKEY(-67) THEN XV=XV+.5:
    F=F-1
2Ø4Ø RETURN
3ØØØ GCOLØ,Ø:MOVEF+2ØØ,96Ø:
    DRAWF+2ØØ,992:MOVEF+2Ø4,96Ø:
    DRAWF+2Ø4,992
3Ø1Ø V=YV*4:IF ABS(V)>5ØØ THEN
    V=5ØØ*SGN(V)
3Ø2Ø MOVE2ØØ,93Ø:MOVE2ØØ,95Ø:
    PLOT85,12ØØ,93Ø:PLOT85,12ØØ,95Ø:
    GCOLØ,3:MOVEV+7ØØ,93Ø:
    MOVEV+7ØØ,95Ø:PLOT85,7ØØ,93Ø:
    PLOT85,7ØØ,95Ø:GCOL3,3
3Ø3Ø RETURN
```

Tandy owners should change the 223s in Lines 2Ø1Ø to 2Ø3Ø to 247.

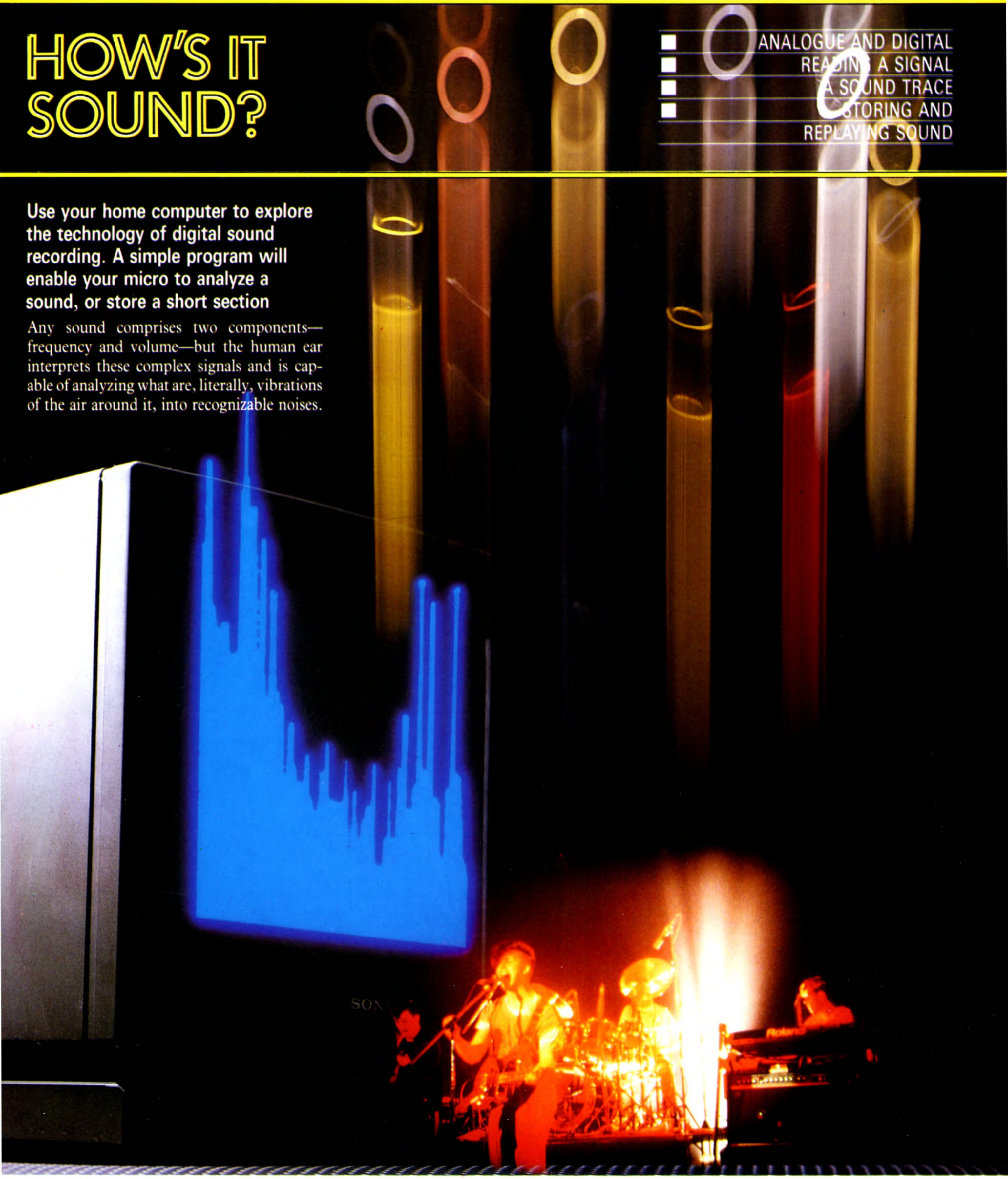
```
1Ø PMODE4,1
2Ø SS=PEEK(186)*256:DIML(1),B(1)
3Ø FORK=ØTO7:READA:POKEK*32+
    SS,A:NEXT
4Ø GET(Ø,Ø)-(7,7),L,G
5Ø DATA 24,6Ø,9Ø,126,126,9Ø,129,129
6Ø PCLS:SCREEN1,1
7Ø DRAW"BMØ,191":FORX=ØTO256
    STEP16:READA:LINE-(X,A),PSET:NEXT:
    PAINT(127,191)
8Ø DATA 151,173,177,165,146,12Ø,167,174,
    177,181,181,17Ø,14Ø,122,158,17Ø,161
9Ø DRAW"BM1,1S8NRDNRDBDNRDRDL"
1ØØ LINE(8,1)-(254,5),PSET,BF:
    LINE(132,7)-(132,12),PSET
11Ø LX=RND(248)-1:LY=15+RND(1Ø):
    XV=RND(15)-8:YV=Ø:F=246
12Ø GOSUB1ØØØ:GOSUB2ØØØ:GOSUB3ØØØ
13Ø IF LY<174 THEN12Ø
14Ø CLS:IF LX<144 OR LX>153
    ORYV>4THEN16Ø
15Ø PLAY"T1ØO4AGFEGFE":PRINT@225,
    "CONGRATULATIONS A SUCCESSFUL
    □□□□LANDING !!":GOTO17Ø
16Ø PLAY"T1ØØO2ADEFGBCDEFA":
    PUT(LX,LY)-(LX+6,LY+7),L,PSET
17Ø FORG=1TO4ØØØ:NEXT
18Ø END
1ØØØ IFLY>12 THENPUT(LX,LY)-
    (LX+7,LY+7),B,PSET
1Ø1Ø LX=LX+XV:LY=LY+YV:S=255-
    (255ANDLY):SOUNDS-(S=Ø),1
1Ø2Ø IF LY<13 THENRETURN
1Ø3Ø IF LX<Ø OR LX>247 THENLX=
    -247*(LX<Ø)
1Ø4Ø GET(LX,LY)-(LX+7,LY+7),B,G
1Ø5Ø PUT(LX,LY)-(LX+7,LY+7),L,
    PSET:RETURN
2ØØØ YV=YV+.5:IF F<1 THENRETURN
2Ø1Ø IFPEEK(341)=223 ANDF>3
    THENYV=YV-1:F=F-3:RETURN
2Ø2Ø IFPEEK(343)=223 THENXV=
    XV-.5:F=F-1:RETURN
2Ø3Ø IFPEEK(344)=223 THENXV=XV+
    .5:F=F-1
2Ø4Ø RETURN
3ØØØ LINE(9+F,1)-(12+F,5),
    PRESET,BF
3Ø1Ø V=2*YV:IFABS(V)>122 THEN
    V=122*SGN(V)
3Ø2Ø LINE(8,8)-(255,11),PRESET,BF:
    LINE(132,8)-(132+V,11),PSET,BF
3Ø3Ø RETURN
```

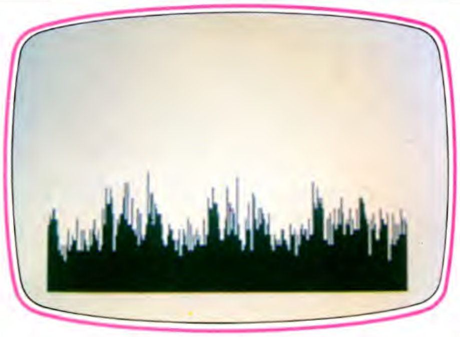
HOW'S IT SOUND?

- ■ ANALOGUE AND DIGITAL
- ■ READING A SIGNAL
- ■ A SOUND TRACE
- ■ STORING AND REPLAYING SOUND

Use your home computer to explore the technology of digital sound recording. A simple program will enable your micro to analyze a sound, or store a short section

Any sound comprises two components—frequency and volume—but the human ear interprets these complex signals and is capable of analyzing what are, literally, vibrations of the air around it, into recognizable noises.





The trace mimics the original sound

In this form, sound is an analogue signal—that is it varies continuously across a wide range, with each variation being significant. Unlike the human senses, computers are not capable of recognizing this sort of change, and instead need a digital signal. In this, each variation is represented by a distinctly different value, either a 1 or Ø, a presence or absence of a signal.

However, although the computer cannot directly interpret a sound signal, it is a fairly simple matter to turn the analogue waveform of a piece of music, say, into digital information for it to use.

In practical applications, this technique is at the forefront of sound recording technology, which is moving from traditional tape storage (of the analogue signal) to computerized systems with disk storage. The advantages are that once you have stored the sound in digital form, it is much easier to modify it or combine it with other sounds—and once recorded, there is far less risk of corruption due to the limitations of the recording system.

MUSIC ON YOUR MICRO

Although such technology at present only exists in a few expensive, highly sophisticated systems, your home computer already has the capability to explore its potential (unless you are an Acorn user). Every time that you LOAD a program stored on cassette tape, you are playing back a signal that has been stored digitally. And, as you will know if you have ever listened to a computer tape, the signal does produce a sound of sorts, too—although being digital, we cannot interpret it meaningfully.

But with suitable programming (which involves using machine code), you can use this system to put music into your micro. All you need to do is to feed an analogue signal into the machine via the cassette socket, and tell the computer how to interpret this—by turning it into a digital signal. (Acorn computers cannot do this without the aid of additional hardware, which is why there are no programs for the BBC or Electron).

Once the micro has done this, it can store the numbers produced in memory—or even display them, as the following sound analyzer program demonstrates. This enables your home micro to translate the analogue signal of any recorded sound into numbers that can be used to produce a graphic trace on the screen, or stored in the memory, for later playback.

SOUND TRACE

The first option the program offers is to produce a continuously changing graphic display of music or sound played on a tape recorder, connected to your micro. When the recorder is switched on, a series of closely spaced lines appear on the screen—the higher the frequency the longer the line. Once the screen is full, the trace will disappear and restart from the left-hand side of the screen.

RECORDING SOUND

The second option allows you to record the sound being fed into the cassette port—although the amount you can store is very limited for the reasons explained below. The third option then allows you to play back the recorded sound.

HOW IT WORKS

As the computer cannot directly interpret the variations of an analogue sound signal, it is programmed to assign digital values to these. What it does is to sample the signal at the cassette port at very rapid intervals—thousands of times per second. The signal at the cassette port can only be high or low, represented by a 1 or a Ø—there are no in-between values as with an analogue signal. But because the sampling is so quick, the rate of change of the digital signals mimics the analogue wave form.

Imagine, for example, that you feed in a signal with a frequency of 256 Hz (the note middle C), then the signal will peak 256 times a second, and each peak lasts for 1/512 seconds. If you read the port 2,ØØØ times a second, then at each peak it will read for approximately four samples. The signal will then read low for four samples, and so on. The variations in the digital values thus mirror the wave form more or less accurately—the more times you sample, the better the computer is able to pick up subtle changes.

The program uses the digital values that it picks up in the two ways described above. To display the waveform graphically, it plots a chart showing how many high bits it finds in a unit time—the overall frequency of that short section of music. Again, the more frequent the sampling, and the more frequent the plotting, the more accurate the display.

Recording is handled in a similar way. The computer counts up eight samples, then stores these as one byte in memory. The process is then repeated for the next eight samples, which are placed in the next memory location. It happens so quickly that this requires a large amount of memory—on the Spectrum, for example, the whole of the available memory is required for about eight seconds of sound.

Playback is just the reverse of this process, with the stored information being directed through the sound output to mimic the vibrations that were present at the cassette port during recording. But because of the limitations of the equipment, the net result is hardly hi-fi!

The Spectrum program comes in two parts—a BASIC program to set up the routines and the screen display, and a machine code routine to read the cassette port. Key in the first part and save it (using SAVE "ANALYZER" LINE 5) then key in the second part which contains the machine code in the form of DATA which is POKEd into memory by Line 3Ø. Line 8Ø contains a checksum to ensure that you have entered the numbers correctly. RUN the program to set up the machine code, then SAVE "ANALYZER" CODE 65368, 1Ø9, immediately after the BASIC set-up program.

LOAD the BASIC program, which will auto-start from Line 5 and load the machine code data into high memory after resetting RAMTOP. When loading is complete, you will be presented with a menu which offers three choices.

The first option will ask you to connect your tape recorder ear socket to the ear socket

Microtip

Setting the sound level

Your normal volume setting may not be ideal for a good sound trace. If your recorder has a PAUSE button, you can use this to set the best level. Run the tape, then PAUSE it while the computer is displaying the trace. You should get a flat trace, showing an unchanging signal. Now turn the tape recorder as high as it will go before background noise starts to disrupt the pattern.

on the Spectrum and then play some recorded sound on the tape recorder. When you subsequently press any key, a continuous frequency versus time graph is generated. To return to the main menu press M, or press F to freeze the display. If you select option F, the display will be regenerated if you then press any key.

If you select option two, you are prompted to press a key to start recording and are notified when the recording has been completed, before an automatic return to the main menu. You can record any sound, but short sharp sounds are best as, due to the limitations of the Spectrum, the sound heard on playback when option three is selected is slower than its natural speed, with most of the low frequency components filtered out—the cassette port cannot distinguish between frequencies below rather a high threshold level. As a result, the recording sounds slurred and deeper in tone.

Due to the huge amount of memory needed to store what is, after all, a small amount of poor quality sound, it's not really worth trying to incorporate the recorded sound into other programs. However, if you want to experiment, it is possible to do this by entering SAVE "sound" CODE 26ØØØ, 3936Ø. Then save the machine code to run the program by entering SAVE "driver" CODE 6544Ø,4Ø.

When you want to use these in your own program, reset RAMTOP by entering CLEAR 25999. Then load the sound data, followed by the driver, by entering LOAD " "CODE : LOAD" " CODE.

To hear the sound, enter RANDOMIZE USR 6544Ø. Remember you will only have about 3K of memory left to write the rest of your program in, which is not a lot unless you are a very efficient machine code programmer.

THE BASIC SET-UP PROGRAM

```
5 CLEAR 25999:LOAD "" CODE
1Ø GOSUB 2ØØ: GOSUB 5ØØ
2Ø IF INKEY$="" THEN GOTO 2Ø
21 LET C=CODE INKEY$: IF C<49 OR
   C>51 THEN GOTO 2Ø
22 GOSUB 3Ø: GOSUB 2ØØ
24 IF C=49 THEN GOTO 1ØØ
25 IF C=5Ø THEN GOTO 6ØØ
26 IF C=51 THEN GOTO 7ØØ
3Ø FOR N=3Ø TO 5Ø STEP 3: BEEP .Ø1,N:
   NEXT N: RETURN
1ØØ CLS : GOSUB 1ØØØ: PRINT AT 12,8;
   BRIGHT 1;"□PRESS ANY KEY□"
1Ø1 IF INKEY$="" THEN GOTO 1Ø1
1Ø2 BEEP .1,1Ø
1Ø4 CLS : GOSUB 15Ø: GOSUB 8ØØ
1Ø5 FOR X=Ø TO 255: PLOT X,Ø: DRAW Ø,
   USR 65368
11Ø IF INKEY$="m" THEN GOSUB 3Ø:
   GOTO 1Ø
111 IF INKEY$="f" THEN GOSUB 8Ø1:
   GOTO 14Ø
13Ø NEXT X: CLS : GOSUB 15Ø: GOSUB 8ØØ:
   GOTO 1Ø5
14Ø PRINT AT Ø,Ø;"□□□□□□□□□
   □□□□□□□□□□□□□□□
   □□□□□□□□": BEEP .1,4Ø:
   PAUSE 5Ø: IF INKEY$="" THEN GOTO
   14Ø
141 BEEP .1,1Ø: CLS : GOTO 1Ø4
15Ø PRINT AT Ø,Ø; BRIGHT 1;
   "□□PRESS (M) TO RETURN TO
   MENU□□": RETURN
2ØØ BORDER 5: PAPER 5: INK Ø: CLS :
   RETURN
5ØØ PRINT AT Ø,5; PAPER 2; INK
   7;"□□SOUND ANALYZER MENU□□"
51Ø PRINT AT 5,7;"1. FREQ. BAR CHART";AT
   7,7;"2. RECORD SOUND";AT 9,7;"3.
   PLAYBACK SOUND"
52Ø PRINT AT 15,5; PAPER 4;" PRESS (1) (2)
   OR (3) ": RETURN
6ØØ PAUSE 2Ø: GOSUB 1ØØØ: PRINT AT
   13,Ø;"Press any key to START
   RECORDING"
6Ø5 IF INKEY$="" THEN GOTO 6Ø5
6Ø6 BEEP .Ø5,2Ø: CLS : PRINT AT
   1Ø,1Ø;"PLEASE WAIT": RANDOMIZE USR
   654Ø8: BEEP .1,3Ø: CLS : PRINT AT 1Ø,6;
   BRIGHT 1;"□RECORDING
   COMPLETE□": PAUSE 3ØØ: GOTO 1Ø
7ØØ RANDOMIZE USR 6544Ø: GOTO 1Ø
8ØØ PRINT AT 1,Ø; PAPER 4;"□□PRESS (F)
   TO FREEZE PICTURE□□": RETURN
8Ø1 PRINT AT 1,Ø; PAPER 4;"□□□PRESS
   ANY KEY TO CONTINUE□□□":
   RETURN
1ØØØ PRINT AT 4,2;"Connect ear socket of
   tape";AT 6,Ø;"recorder to ear socket of
   your";AT 8,Ø;"Spectrum. Then play some
   music.": RETURN
```

THE MACHINE CODE ROUTINE

```
1Ø CLEAR 26ØØØ: RESTORE : LET t=Ø: LET
   x=65368
2Ø FOR n=1 TO 1Ø8
3Ø READ a: POKE x,a
4Ø LET t=t+a
5Ø LET x=x+1
6Ø NEXT n
7Ø IF t=12721 THEN PRINT "OK.": STOP
8Ø PRINT "DATA ERROR": STOP
9Ø DATA 14,64,175,8,17,2Ø8,7,219,254,23Ø,
   64,185,4Ø,7,62,64,169,79,8,6Ø,8,29,32,
   239,175,178,4Ø,5,3Ø,255,21,24,23Ø,8,2Ø3,
   63,6,Ø,79,2Ø1
1ØØ DATA 243,33,144,1Ø1,17,8Ø,255,6,7,219,
   254,2Ø3,119,32,2,2Ø3,254,2Ø3,62,16,244,
   35,125,187,32,237,124,186,32,233,251,
   2Ø1,243,33,144,1Ø1,17,8Ø,255,6,8,2Ø3,7Ø,
   4Ø,4
11Ø DATA 62,Ø,211,254,62,255,211,254,2Ø3,
   14,16,24Ø,35,125,187,32,233,124,186,32,
   229,251,2Ø1
```

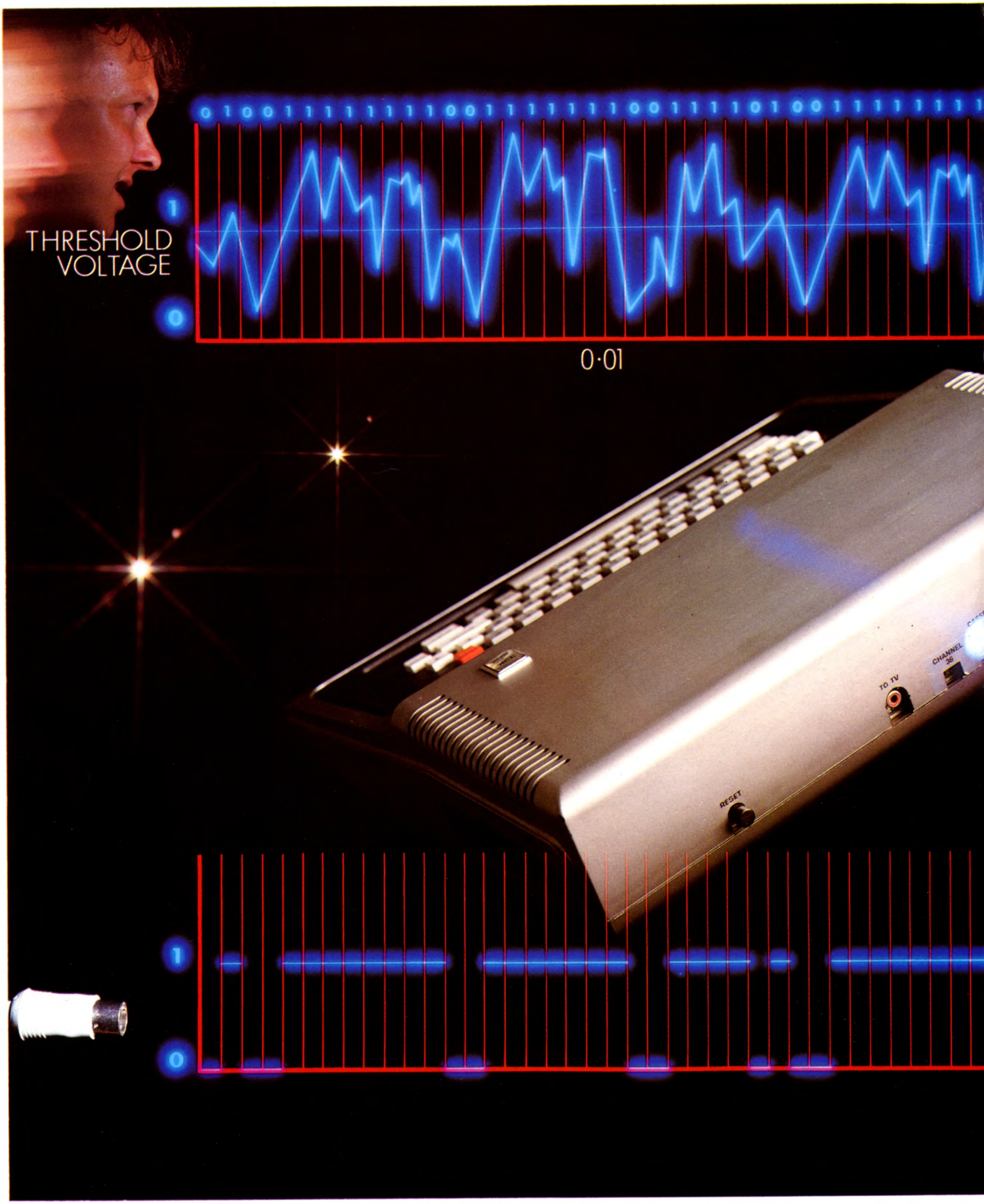
The Commodore program is only capable of fulfilling the first option—it produces a multi-coloured sound trace. It is possible to produce a program which will allow you to store a small amount of sound, but the amount of programming required to output a sound is beyond the scope of this article.

The machine code data for the routine which reads the cassette port is contained within the main program as DATA starting at Line 1ØØØ. Once the program is entered, SAVE to tape or disk before RUNning it. You must then RUN the program. As the routine is so short, the machine code data is stored in the cassette buffer. Once the program is RUN, a message INSERT TAPE & PRESS PLAY THEN HIT RETURN appears on the screen.

If you want to freeze the sound trace at a particular point, press any key. The picture on the screen will remain frozen until you release the depressed key.

```
5 FOR Z=832 TO 852:READ X:POKE Z,X:
   NEXT Z
8 PRINT "♥INSERT TAPE & PRESS PLAY
   THEN HIT RETURN":INPUT A$
1Ø HIRES Ø,1:COLOUR 1,1:MULTI 3,4,7
2Ø FOR Z=Ø TO 159
3Ø SYS 832
4Ø IFPEEK(251)*4<2ØØ THEN LINE Z,2ØØ,Z,
   2ØØ-PEEK(251)*4,RND(1)*3+1:GOTO 5Ø
45 IF PEEK(197)<>64 THEN 45
5Ø NEXT Z:GOTO 1Ø
1ØØØ DATA 162,Ø,134,251,173,13,22Ø,197,
   252,24Ø,6,133,252,23Ø,251,24Ø,3,232
1Ø1Ø DATA 2Ø8,24Ø,96
```

The machine code data for the routine which reads the cassette port is contained within the main program as DATA starting at Line 1ØØØ. A checksum is contained at the end of each line to enable you to check that you have entered the data correctly—when you run it the line will throw up an error report if there is a mistake. To use the program with the Tandy, certain alterations have to be made. In Line 1Ø1Ø, substitute 18Ø for 14Ø and 244 for 55. The latter numbers are printed in bold so you can easily identify them. Checksum 1838 then becomes 2Ø67. In Line 1Ø2Ø make the same substitutions and change the checksum



0 1 0 0 1 1 1 1 1 1 1 1 0 0 1 1 1 1 1 1 1 0 0 1 1 1 1 0 1 0 0 1 1 1 1 1 1 1 1
1
THRESHOLD VOLTAGE
0
0·01
TO TV
CHANNEL 36
RESET
1
0

An analogue signal coming into the cassette port is sampled 2000 times a second. When the signal is above the threshold voltage, it is detected and registers a 1. Signals below this level register a 0. The digitally-produced trace thus, in effect, mimics the analogue sound wave

from 1618 to 1847. Also change USRØ1 in Line 42Ø to USR1 and similarly in Line 61Ø change USRØ2 to USR2.

Once the program is entered, SAVE to tape or disk before RUNning it. You must then RUN the program to POKE the machine code into higher memory. Once the program is RUN, the main menu will appear on the screen. If there is a mistake in the DATA, the message CHECK DATA IN LINE (relevant number) will appear on the screen.

Press 1 for the first option. You will then be asked to position the tape. Press play on the recorder and ENTER, when the sound trace will automatically start. Press M to return to the main menu or SHIFT @ to freeze the picture.

Press 2 for the second option (recorded sound), and you will be given the same instructions as for option 1. While the music is being recorded, a coloured display will appear on the screen. Once recording has finished, the display will disappear and the program returns to the main menu.

Press 3 to play back the music you have just recorded. Unlike that on the Spectrum, music recorded on the Dragon or Tandy will play back at the same speed as the original.

If you want to use the recorded sound in your own program, you can store it by entering CSAVEM"MUSIC",1536,13823,1536 (add 1536 to these three values if you are using a disk drive). To save the machine code which permits the program to run, enter CSAVEM "DRIVER", 31ØØØ, 31174, 31Ø91. When you want to use this, reset RAMTOP by entering CLEAR 2ØØ, 3Ø999 and use PCLEAR 8 to protect the DATA. To load sound data, enter CLOADM "MUSIC" and for the machine code CLOADM "DRIVER". To play the recorded sound, enter EXEC 31Ø91.

If you do this, bear in mind that the recorded sound takes up a large amount of memory, leaving less space for your own program.

```
10 PCLEAR8:CLEAR2ØØ,3Ø999:CLS:B=191
2Ø K=31ØØØ
3Ø READA:IF A<Ø THEN6Ø
4Ø IF A<256 THENPOKEK,A:K=K+1:T=
   T+A:GOTO3Ø
5Ø IF T<>A THENPRINT" error.CHECK
   DATA IN LINE";1ØØØ+1Ø*INT
   ((K-31ØØ1)/2Ø):END ELSET=Ø:GOTO3Ø
6Ø DEFUSRØ=31ØØØ:DEFUSR1=31Ø44:
   DEFUSR2=31Ø91
7Ø PRINT@14,"menu":PRINT@
   135,"1-FREQ. BAR CHART":PRINT@
   199,"2- RECORD SOUND":PRINT@263,
   "3- PLAYBACK SOUND"
8Ø A$=INKEY$:IF A$<"1" OR A$>"3"
   THEN8Ø
9Ø ON VAL(A$) GOSUB2ØØ,4ØØ,6ØØ
1ØØ CLS:GOTO7Ø
2ØØ PMODE4:COLORØ,5:CLS
21Ø PRINT" POSITION TAPE, PRESS PLAY
    □□□□□□□FOLLOWED BY enter"
22Ø MOTORON:AUDIOON
23Ø IFINKEY$<>CHR$(13) THEN23Ø
24Ø MOTOROFF:PRINT@192,"PRESS 'M'
    FOR MENU OR [SHIFT] @ TO FREEZE
    PICTURE"
25Ø FORG=1TO4ØØØ:NEXT:MOTORON
26Ø SCREEN1,1
27Ø PCLS:FORX=ØTO255:A=B-4*
    USRØ(Ø):IF A<Ø THENA=Ø
28Ø LINE(X,B)-(X,A),PSET
29Ø IF INKEY$="M" THENX=255:NEXT:
    MOTOROFF:RETURN
3ØØ NEXT:GOTO27Ø
4ØØ CLS:PMODE3:MOTORON:AUDIOON:
    PRINT" POSITION TAPE, THEN PRESS
    PLAY FOLLOWED BY enter"
41Ø IF INKEY$<>CHR$(13) THEN41Ø
42Ø SCREEN1,Ø:N=USRØ1(Ø)
43Ø MOTOROFF:RETURN
6ØØ CLS:PRINT" PLAYING BACK RECORDED
    SOUND"
61Ø N=USRØ2(Ø)
62Ø PRINT@129,"AGAIN (Y/N) ?"
63Ø A$=INKEY$:IF A$<>"Y" AND
    A$<>"N" THEN63Ø
64Ø IF A$="Y" THEN 6ØØ
65Ø RETURN
1ØØØ DATA 26,8Ø,2Ø6,255,32,142,2,233,2Ø4,
     Ø,Ø,1Ø2,196,37,1Ø,16,163,132,48,31,1915
1Ø1Ø DATA 38,245,126,140,55,195,Ø,1,32,7,
     1Ø2,196,36,237,16,163,132,48,31,38,1838
1Ø2Ø DATA 245,126,140,55,26,8Ø,142,Ø,Ø,48,
     31,38,252,22Ø,25,131,Ø,1,52,6,1618
1Ø3Ø DATA 158,186,198,8,134,11,74,38,253,
     118,255,32,1Ø5,132,9Ø,39,4,18,18,32,19Ø3
1Ø4Ø DATA 4,48,1,198,8,172,228,38,231,53,
     134,26,8Ø,182,255,1,132,247,183,255,2476
1Ø5Ø DATA 1,182,255,3,132,247,183,255,3,
     182,255,35,138,8,183,255,35,158,186,22Ø,
     2916
1Ø6Ø DATA 25,131,Ø,1,52,6,134,8,52,2,23Ø,
     128,88,36,4,134,252,32,3,79,1397
1Ø7Ø DATA 33,253,183,255,32,134,8,74,38,
     253,33,251,1Ø6,228,39,8,1Ø9,159,31,64,
     2291
1Ø8Ø DATA 3Ø,136,32,224,134,8,167,228,172,
     97,38,214,53,146,1679,-1
```

THE GAME OF FOX AND GEESE

Your computer can be a cunning Fox and Geese player. See how Artificial Intelligence methods can be applied to writing games programs which can think ahead

Earlier in *INPUT* you saw how an Othello program could be written in which the computer could play an acceptable game against a human player. However, with a little practice, you should be able to beat the computer most of the time.

Over the next three parts of Games Programming you'll see how a more sophisticated program can be written, to enable your computer to play the game of Fox and Geese. The program has many levels of skill, so you can make the game as difficult or as easy as you wish.

Fox and Geese illustrates many of the problems and principles involved in writing one of the most interesting and enduring of games programs to play—namely, chess. An average commercial chess program running on a home micro can give most chess players a run for their money, unless they are of a very good standard, or understand the weaknesses of computer chess.

It's impossible to write a chess program that will be worth playing in BASIC, because the machine will take an age over each move. In the higher levels of Fox and Geese, you'll find that the program takes rather a long time considering each move—perhaps in excess of half an hour on the very highest levels running on the slower machines. The problems would increase many-fold if you were to attempt to write a chess-playing program instead.

To keep the exercise out of the realms of machine code, a simpler game is needed. Fox and Geese fills the bill very nicely because it has many of the characteristics of chess, and even takes place on the same kind of board.

This article is in three parts. In this first part, you will see the principles behind a program of this type—building up to a complete, playable game in parts two and three. But let's start by looking at the game itself, and what is required of the program to play it.

THE GAME

Fox and Geese is played on the white squares of a standard chessboard. There is one fox which starts off at one end of the board, and four geese, which start from the opposite end. One player controls the fox, and the other the geese. The object of the game is for the fox to find its way past the geese and reach the opposite end of the board, or for the geese to corner the fox.

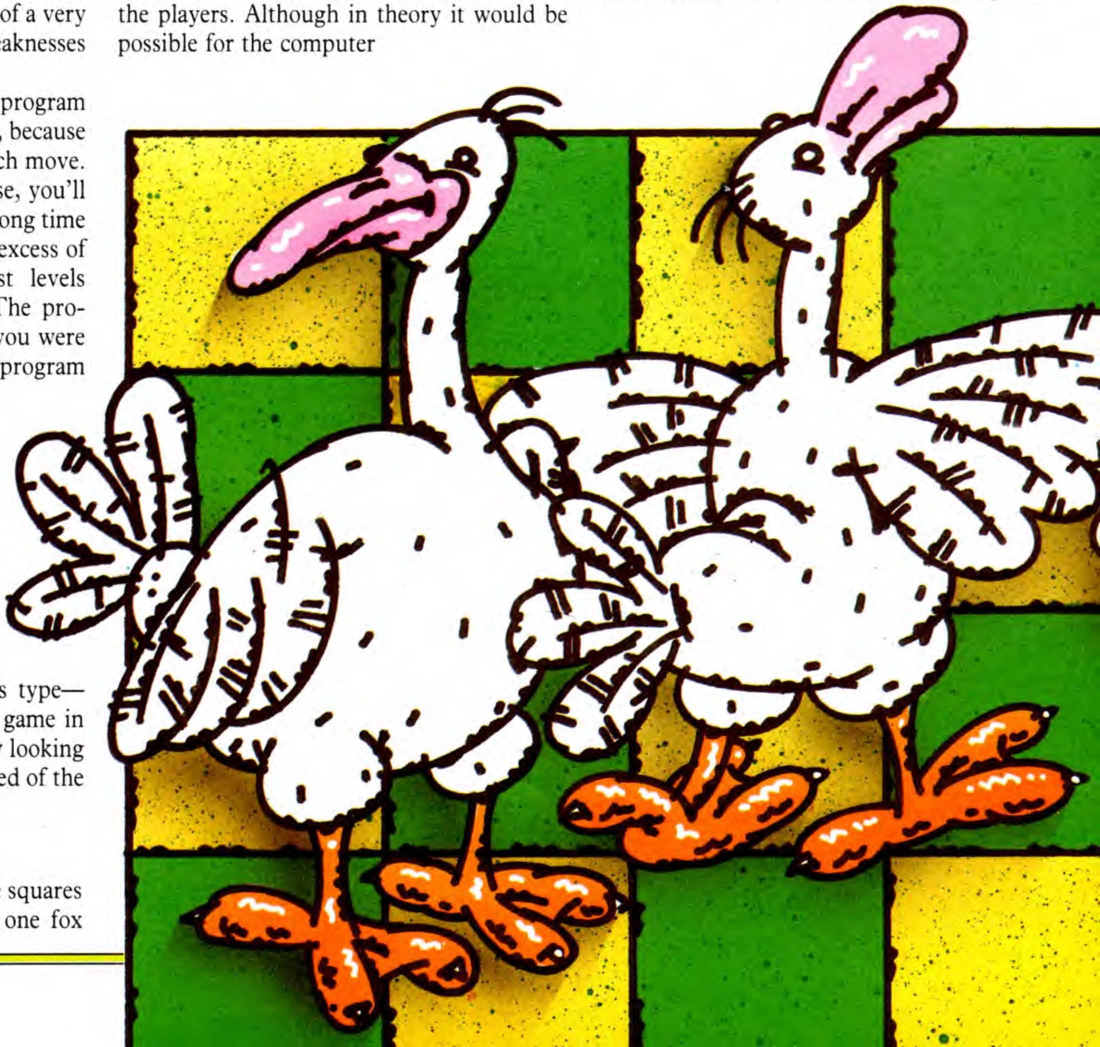
With four against one, the game may seem a little one-sided, but the geese are limited to forward movement only, whereas the fox may move backwards as well as forwards. The program has been written so that the computer may play either the fox or the geese, or may be set to play against itself.

TACKLING THE PROBLEMS

Fox and Geese is very like chess in that there is no element of chance within the game—the outcome is totally dependent on the skill of the players. Although in theory it would be possible for the computer to learn to play through trial and error, like a human player, at the current state of the art this isn't the best way of solving the problems involved—and the program wouldn't fit in your micro!

Programming a game like Fox and Geese, or chess, is really an exercise in Artificial Intelligence. To offer a stern challenge to a human opponent, the computer must be able to play intelligently. Unfortunately, the machine cannot look at a board and absorb the spatial relationships between the pieces the way you can. Instead, the positions on the board have to be converted to numeric values that the computer can understand.

If you want to write a program that will enable a computer to play intelligently, you must first look at the nature of the game. The type of game will dictate the type of program.

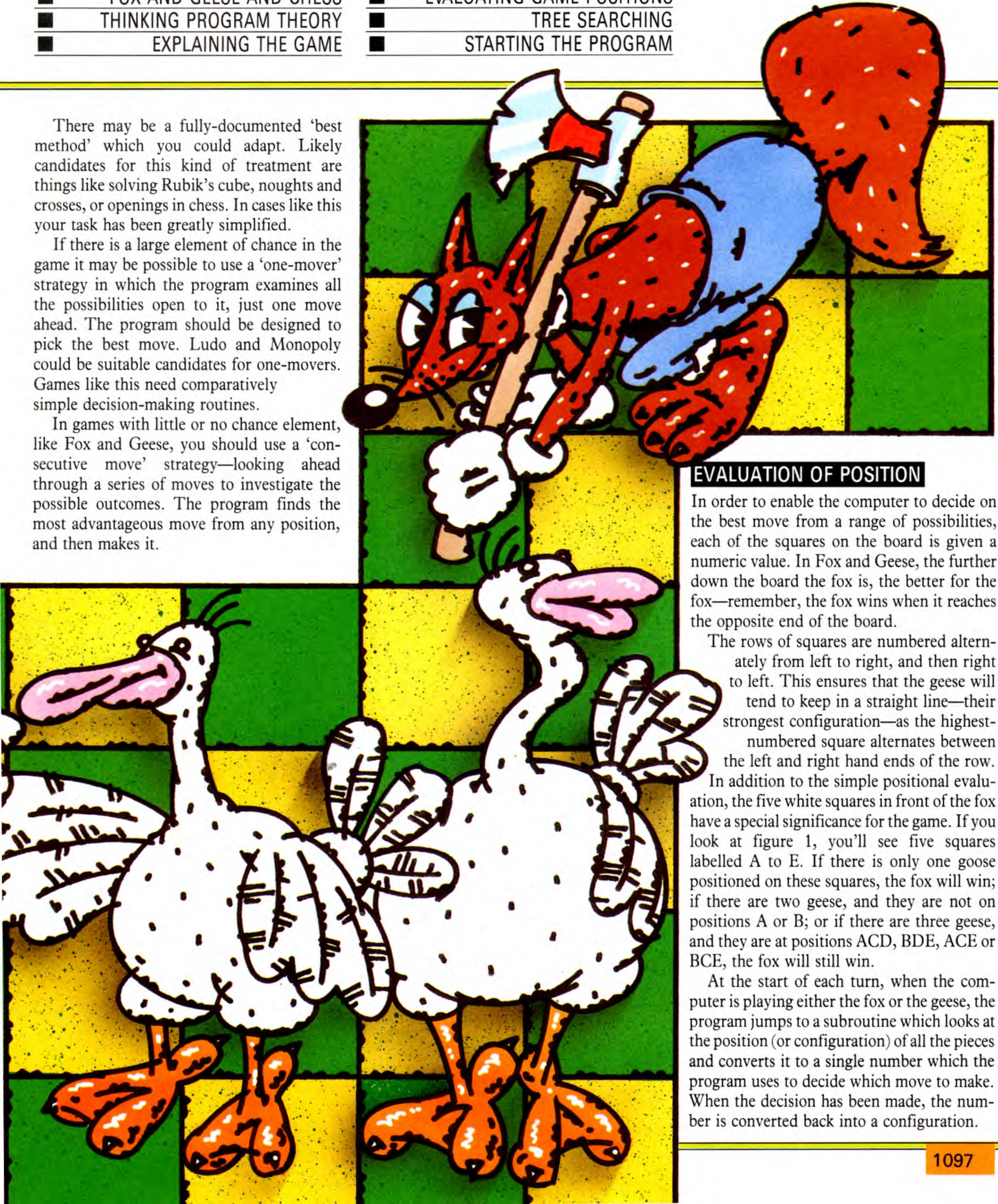


- MAKING THE COMPUTER A SKILFUL PLAYER
- FOX AND GEESE AND CHESS
- THINKING PROGRAM THEORY
- EXPLAINING THE GAME
- TACKLING PROGRAMMING PROBLEMS
- EVALUATING GAME POSITIONS
- TREE SEARCHING
- STARTING THE PROGRAM

There may be a fully-documented 'best method' which you could adapt. Likely candidates for this kind of treatment are things like solving Rubik's cube, noughts and crosses, or openings in chess. In cases like this your task has been greatly simplified.

If there is a large element of chance in the game it may be possible to use a 'one-mover' strategy in which the program examines all the possibilities open to it, just one move ahead. The program should be designed to pick the best move. Ludo and Monopoly could be suitable candidates for one-movers. Games like this need comparatively simple decision-making routines.

In games with little or no chance element, like Fox and Geese, you should use a 'consecutive move' strategy—looking ahead through a series of moves to investigate the possible outcomes. The program finds the most advantageous move from any position, and then makes it.



EVALUATION OF POSITION

In order to enable the computer to decide on the best move from a range of possibilities, each of the squares on the board is given a numeric value. In Fox and Geese, the further down the board the fox is, the better for the fox—remember, the fox wins when it reaches the opposite end of the board.

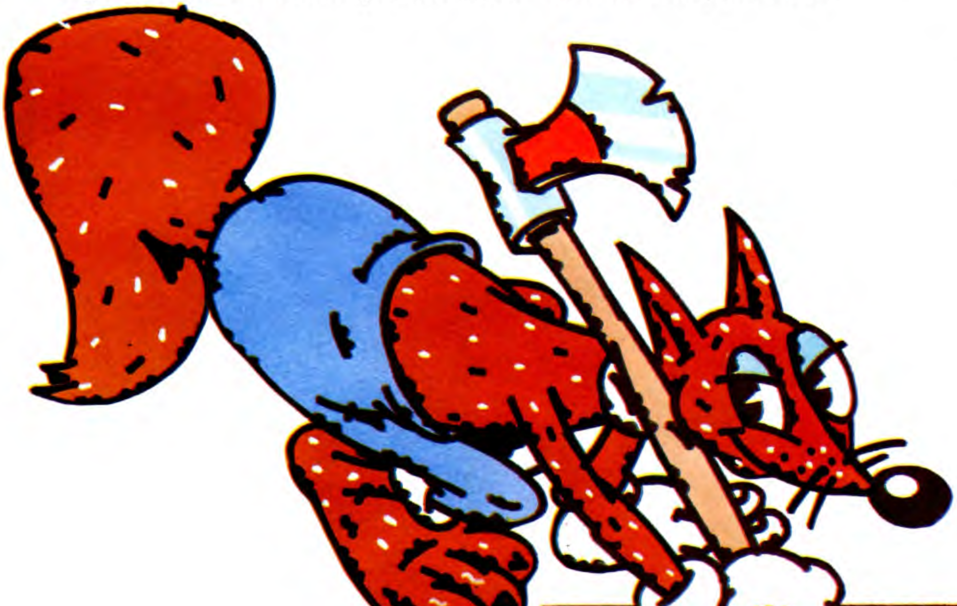
The rows of squares are numbered alternately from left to right, and then right to left. This ensures that the geese will tend to keep in a straight line—their strongest configuration—as the highest-numbered square alternates between the left and right hand ends of the row.

In addition to the simple positional evaluation, the five white squares in front of the fox have a special significance for the game. If you look at figure 1, you'll see five squares labelled A to E. If there is only one goose positioned on these squares, the fox will win; if there are two geese, and they are not on positions A or B; or if there are three geese, and they are at positions ACD, BDE, ACE or BCE, the fox will still win.

At the start of each turn, when the computer is playing either the fox or the geese, the program jumps to a subroutine which looks at the position (or configuration) of all the pieces and converts it to a single number which the program uses to decide which move to make. When the decision has been made, the number is converted back into a configuration.

TREE SEARCHING

The possible moves from a position on the board can be represented by a tree structure, with branches emanating from the piece's (root) position. For example, from the position in figure 2, the first level of the tree in figure 3 could be drawn. If you look ahead two moves, the tree becomes more complex, as in the second level—notice that there is no reason for there always to be four branches from each square, because the piece may be blocked by another, or may find itself at the outside edge of the board.



SPEEDING UP THE PROGRAM

With the number of calculations to be performed, BASIC can prove very slow, but there are three ways to speed up the program. Firstly, you can make sure that the program doesn't bother to evaluate the opponent's next move, if your move has won the game. However, this only saves time at the end, not throughout the game.

Secondly, the same configuration can very frequently be reached from many different routes. If this is the case, it may pay you to build up a table of common configurations and the calculated values associated with them. Such a table stops the program assessing the position again individually each time, but there is no doubt in storing a configuration which would take less time to evaluate than it would to find it in the table. Theoretically, the table can be used only when the program is looking ahead three *plies* (complete turns of both players) or more, the practice of the game proves that there is no point in storing anything in a table unless at least five plies are being looked at.

Thirdly, the so-called *alpha-beta algorithm* may be employed. This was discovered in the early 1960s by researchers in the field of artificial intelligence, and comes into play every time the tree search needs to examine more than one ply of a tree.

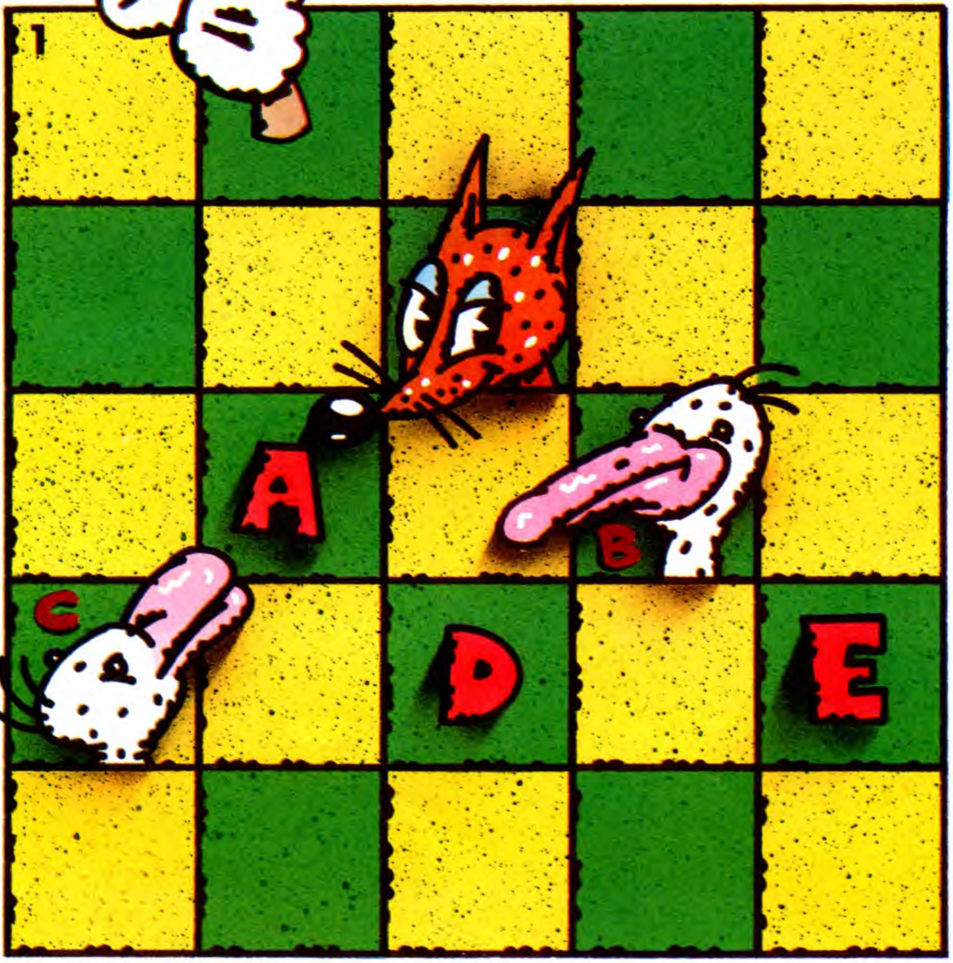
If you look at figure 3, you can see the evaluations of the possible moves from a fox position. The program will fully evaluate branch A, and then pass on to branch B. If, at any stage in evaluating B, a worse outcome is found, all the moves in B will be rejected. The best move so far is remembered by the program and compared with the outcomes in each branch in turn. Finding at *any* point a worse outcome will cause the *whole* branch to be rejected.

The alpha-beta algorithm really comes into its own as the tree gets more and more complex. If the tree gets big enough, it can allow you to discard about 99.8% of the possibilities at an earlier stage, with a similar saving in time. But in this game the tree will not approach this level of complexity.

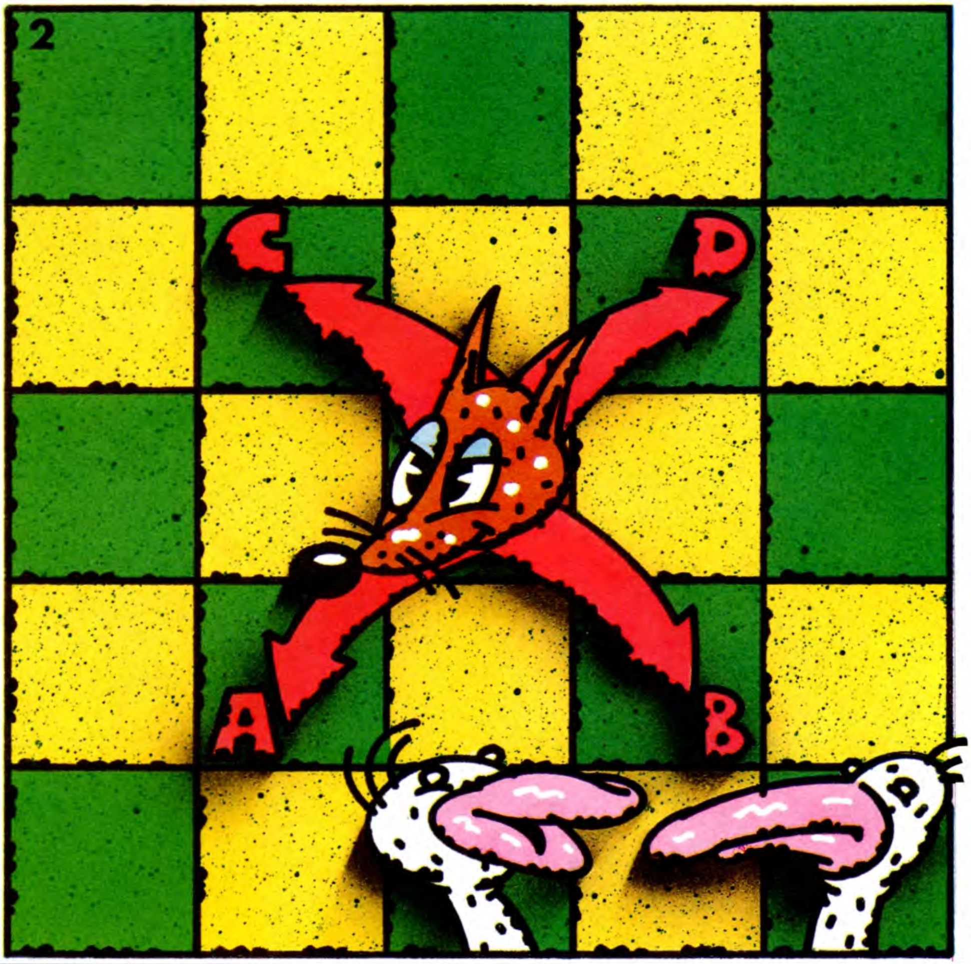
STARTING THE PROGRAM

Now that you have some idea of the theory behind writing this kind of game, you can pass on to entering the first section of program. It concerns setting up the graphics, but you won't see anything at this stage if you RUN the program. Don't forget to SAVE it on tape ready for the next part of the program.

1



```
10 DEF FN A(F)=INT (LN (F)/L2+.ØØ1)
100 GOTO 2ØØ2
2ØØ2 GOSUB 5ØØØ
2ØØ6 BORDER 4: PAPER 4: CLS
2ØØ8 PRINT AT 8,1; FLASH 1;“PLEASE WAIT
   WHILE SETTING UP”
5ØØØ FOR J=USR “A” TO USR “D”+7:
   READ A: POKE J,A: NEXT J: RETURN
5Ø7Ø DATA 2Ø,28,55,127,15,2Ø,4Ø,72,Ø,Ø,
   248,252,25Ø,4Ø,2Ø,2Ø,Ø,Ø,Ø,7,2Ø5,123,6Ø,
   15,12,2Ø,31,152,248,216,48,224
6ØØØ LET S$(1)=“■■□1■■□2■
   ■□3■■□4”
6Ø1Ø LET S$(2)=“□8■■□7■■□6
   ■■□5■■”
6Ø15 LET S$(3)=“■■□9■■1Ø■■
   11■■12”
6Ø3Ø LET S$(4)=“16■■15■■14■
   ■13■■”
6Ø4Ø LET S$(5)=“■■17■■18■■
   19■■2Ø”
6Ø5Ø LET S$(6)=“24■■23■■22■
   ■21■■”
6Ø6Ø LET S$(7)=“■■25■■26■■
   27■■28”
6Ø7Ø LET S$(8)=“32■■31■■3Ø■
   ■29■■”
61ØØ RETURN
```

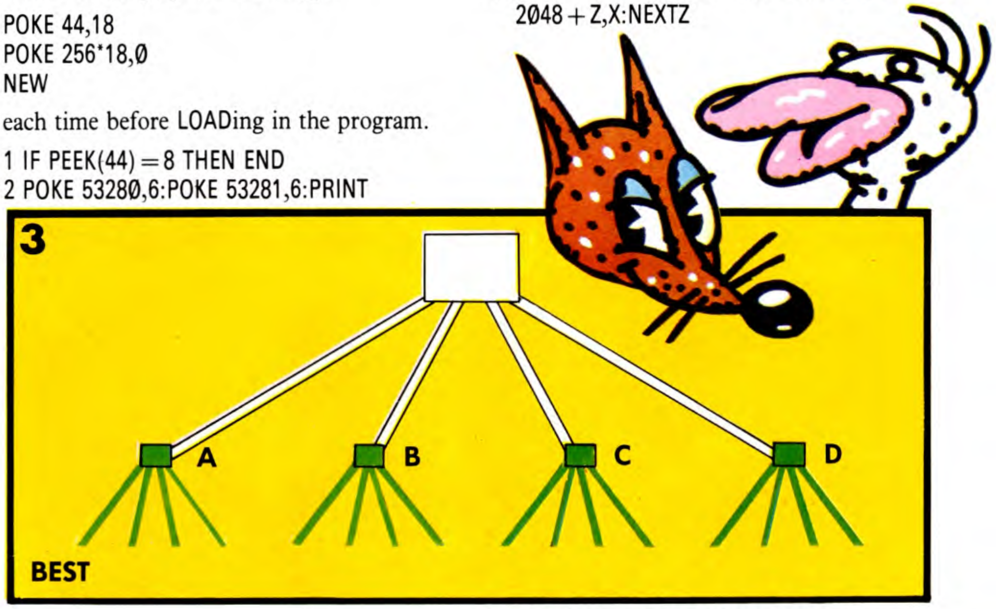


Before using the Commodore program you must move up the start of BASIC to make room for the graphics sets. Type

```
POKE 44,18
POKE 256*18,Ø
NEW
```

each time before LOADing in the program.

```
1 IF PEEK(44)=8 THEN END
2 POKE 5328Ø,6:POKE 53281,6:PRINT
   “♥π DEFINING GRAPHICS...”:
   GOSUB 4ØØØ
4ØØØ POKE 56334,Ø:POKE 1,35
4Ø1Ø FOR Z=Ø TO 1Ø23:POKE 2Ø48+Z,
   PEEK(53248+Z):NEXTZ
4Ø2Ø POKE 1,39:POKE 56334,1
4Ø3Ø FOR Z=68Ø TO 711:READ X:POKE
   2Ø48+Z,X:NEXTZ
```

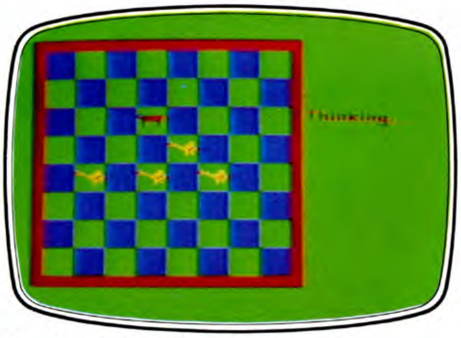


```
4Ø4Ø FOR Z=3329 TO 3334:POKE Z,129:
   NEXTZ:POKE 3328,255:POKE 3335,255
4Ø5Ø RETURN
5ØØØ DATA Ø,Ø,Ø,7,2Ø5,123,6Ø,15,12,2Ø,31,
   152,248,216,48,224
5Ø1Ø DATA 2Ø,28,55,127,15,2Ø,4Ø,72,Ø,Ø,
   248,252,25Ø,4Ø,2Ø,2Ø
```

```
2 MODE 4
1Ø PROCCHARS
4ØØØ DEFPROCCHARS
4Ø1Ø REM 224:225 226:227
4Ø2Ø VDU 23,224,Ø,Ø,Ø,7,2Ø5,123,6Ø,15
4Ø3Ø VDU 23,225,12,2Ø,31,152,248,216,48,224
4Ø4Ø VDU 23,226,2Ø,28,55,127,15,2Ø,4Ø,72
4Ø5Ø VDU 23,227,Ø,Ø,248,252,25Ø,4Ø,2Ø,2Ø
4Ø6Ø VDU 23,228,128,128,128,128,128,128,
   128,255
4Ø7Ø VDU 23,229,1,1,1,1,1,1,1,255
4Ø8Ø VDU 23,23Ø,255,128,128,128,128,128,
   128,128
4Ø9Ø VDU 23,231,255,1,1,1,1,1,1,1
41ØØ VDU 23,232,8Ø,16Ø,8Ø,16Ø,8Ø,16Ø,8Ø,
   16Ø
411Ø VDU 23,233,255,255,255,255,255,255,
   255,255
5ØØØ ENDPROC
```

```
5 CLS:PRINT@23Ø,“SETTING UP
   GRAPHICS”:GOSUB4ØØØ
1Ø SCREEN1,Ø:GOTO1Ø
```

```
4ØØØ PMODE3:PCLS:DIMGS(5),FX(4),SQ(1Ø)
4Ø1Ø DRAW"BM3,ØC2FGR3FR5E3RED3FDGL
   GNFLNG2LH2LH":PAINT(12,5),2:PSET
   (Ø,1,4):PSET(14,5,4):PSET(16,4,4):GET
   (Ø,Ø)—(19,9),GS,G
4Ø2Ø DRAW"BM18,2ØC4GL13HLG2R2F2ND4
   R1ØND4U2":PAINT(1Ø,22),4:PSET(2,21,1):
   GET(Ø,2Ø)—(19,28),FX,G
4Ø3Ø LINE(Ø,Ø)—(175,175),PSET,BF:COLOR
   3:LINE(8,8)—(167,167),PSET,BF
4Ø4Ø FORK=8TO128 STEP4Ø:FORL=28TO
   148 STEP4Ø:PUT(L,K)—(L+19,K+19),
   SQ,PSET:PUT(L—2Ø,K+2Ø)—(L—1,
   K+39),SQ,PSET:NEXTL,K
4Ø5Ø PUT(68,13)—(87,21),FX,PSET
4Ø6Ø FORK=8TO128 STEP4Ø:PUT(K,153)—
```



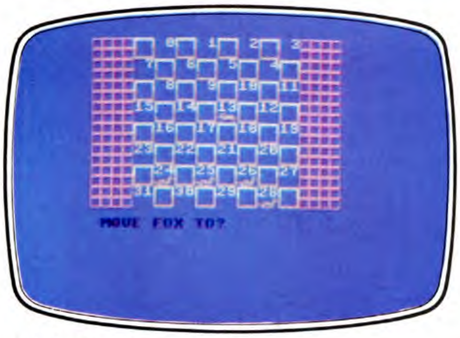
The Dragon's colourful board ...

```
   (K+19,162),GS,PSET:NEXT
4Ø7Ø TH$="R2ND6R2BR4D6BR4U3LBR6ND
   3BU2UBF3ND3R4D3BR5U6D3NR3F3BR4U
   3BU2UBF3BR2ND3R4D3BR7L3U3R3D6L3
   BE3BR4RBR5RBR5R"
4Ø8Ø MW$="ND6F3RU3D6BR9L4U3R4D3B
   E3F3UE2BR3R3DL3D2R3BR6NU6E2F2NU6
   BR3U6D3R4D3BR7L3U3R3DL3BR7NUNR2
   D2BR8L3U3R3DL3BE4BR7R4D3L4D3"
4Ø9Ø WG$="RD6E2F2U6BR4D6BR4U3NL2B
   F3U3BU2UBF3BRNR3D3R3BR4NU6BR4U3
   NL3BE3BR8L4D6R4U3BR4D3R4U3L4BR8
   D3R4U3L3BR9BUL2D2R2D2L2BR9L3U3R3
   DL"
41ØØ V$="T4O2DEFBGACDEGGDCDE"
42ØØ C=1:G=Ø:RETURN
5ØØØ FORK=1TO14:PUT(2ØØ,5)—(21Ø,15),
   SQ,PRESET:PLAY"T5ØAC":PUT(2ØØ,5)—
   (21Ø,15),SQ,PSET:PLAY"DA":NEXT:
   RETURN
```

DISPLAYING THE BOARD

```
31Ø LET F=FN A(ABS (P))—3Ø: LET
   B=P/B(F): IF B<Ø THEN LET
   B=B+BX: LET F=33—F
32Ø LET C=B/B(29): FOR A=8 TO 1 STEP
   —1: LET R$(A)=B$(INT (C)+1,(2—FN
   W(A))): LET C=(C—INT (C))*16: NEXT A
33Ø LET R$(INT (F/4+.8))(FN C(F)+1 TO
   FN C(F)+4)=F$(FN W(F/4—.2)+1)
34Ø FOR A=1 TO 8: PRINT AT 2*A,8; PAPER
   7;S$(A): PRINT AT 2*A+1,8; PAPER
   7;R$(A): NEXT A: RETURN
```

```
31Ø F=FNA(ABS(P))—31:B=P/B(F):IF
   B<ØTHENB=B+BX:F=31—F
32Ø C=B/B(28):FORA=7TOØSTEP—1:R$
   (A)=B$(INT(C),1ANDA):C=(C—INT
   (C))*16:NEXTA
33Ø R$(F/4)=LEFT$(R$(F/4),FNC(F))+F$
   (F/4AND1)+RIGHT$(R$(F/4),12—FNC
   (F))
34Ø PRINT"♥";:FORA=ØTO7:PRINT TAB
   (7);"▟▩□□□□▄◣";S$(A)"▟
   ▩□□□□"
35Ø PRINT TAB(7);"▟▩□□□□
   ▄♣";R$(A)"▟▩□□□□":
   NEXTA:PRINT"■":RETURN
```

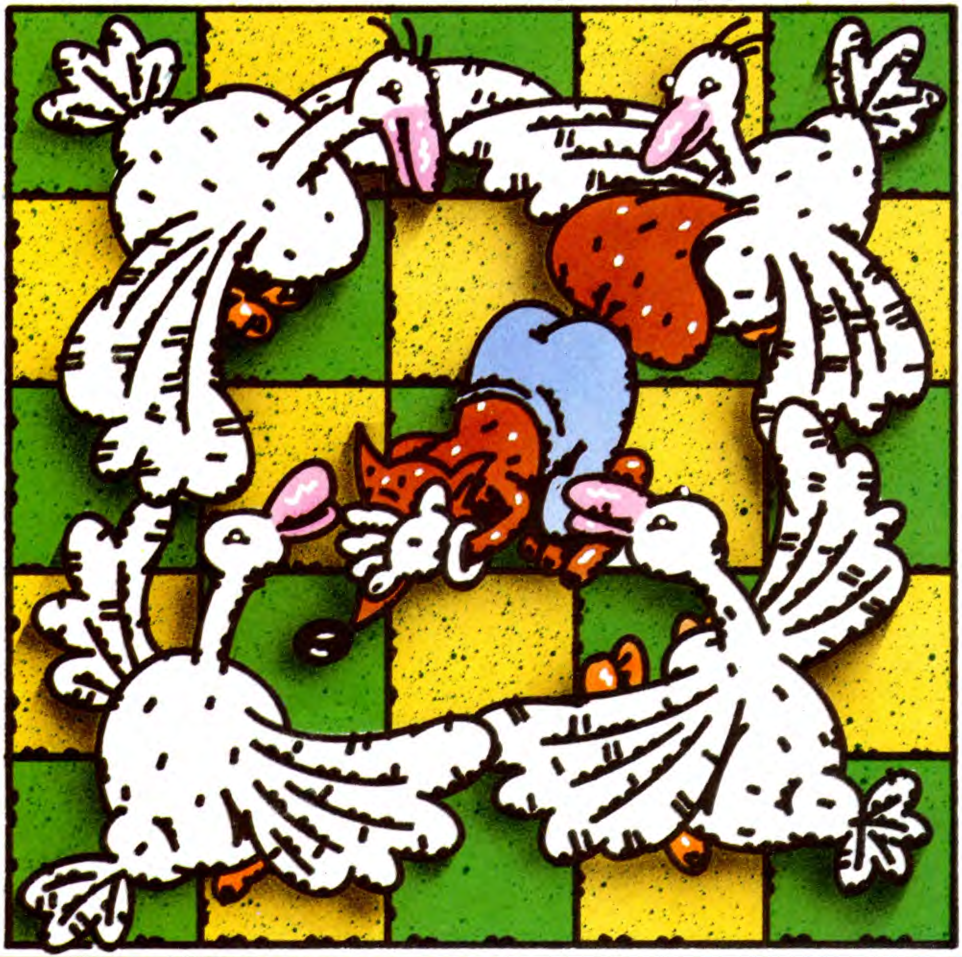


and the Acorn version

```
31Ø F=FNA (ABS(P))— 31: B=P/ B(F):
   IFB<Ø THENB=B+BX:F=31—F
32Ø C=B/B(28): FOR A=7TOØ STEP—1:
   R$(A)= B$ (INT(C),1 AND A):C=(C—
   INT(C))*16: NEXTA
33Ø R$(F/4)= LEFT$ (R$(F/4), FNC
   (F))+F$(F/4 AND 1)+RIGHT$
   (R$(F/4), 12— FNC (F))
34Ø CLS: FOR A=ØTO7: PRINT TAB(8)
   CHR$ 233; CHR$ 233; CHR$ 233;
   CHR$ 233; S$(A) CHR$ 233; CHR$ 233;
   CHR$ 233; CHR$ 233
35Ø PRINT TAB(8) CHR$ 233; CHR$ 233;
   CHR$ 233; CHR$ 233; R$(A) CHR$ 233;
   CHR$ 233; CHR$ 233; CHR$ 233:NEXTA:
   RETURN
```

Lines 31Ø to 35Ø display the board with the five pieces in position. The subroutine is called once a turn, by fox and geese alike.

The drawing instructions for the Dragon and Tandy are in the subroutine along with the high resolution graphics DATA.



CLIFFHANGER: STARTING OFF

The 'CLIFFHANGER' listings published in this magazine and subsequent parts bear absolutely no resemblance to, and are in no way associated with, the computer game called 'CLIFF HANGER' released for the Commodore 64 and published by New Generation Software Limited.

You've got a title page, an instruction page, cliff and a tune. But you need a routine that tells them when they're needed and starts Willie off

This is part seven of Cliffhanger, so you might think that you have already started the game.

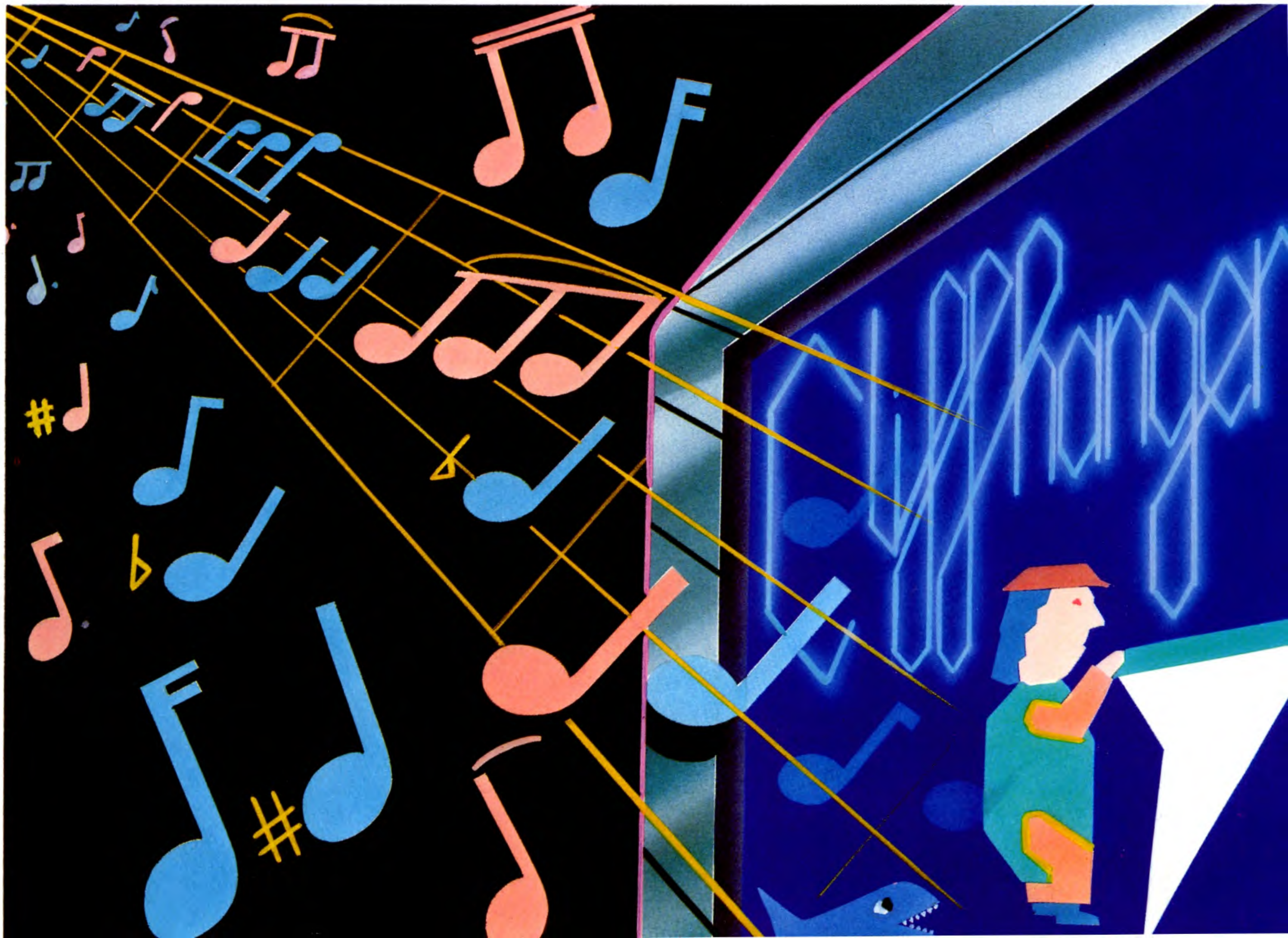
A machine code game needs an initialization routine which calls the titles and the instruction and the music and sets the score, levels and lives to their starting values, and starts off the game. Its origin is the location you call when you want to run the program. This is called from BASIC and normally you'd have a bootstrap program (see pages 459 to 463). Here, though, there is no real need for protection. Besides, you need to be able to get inside the program to modify it—so you won't want it auto-running.

This is it: the routine that starts off the Spectrum version of Cliffhanger.

```
        org 58576
gbin    call ti
        ld a,5
        ld (57343),a
        ld a,Ø
        ld (57344),a
        ld hl,Ø
        ld (57337),hl
        ld (57339),hl
        ld (57341),hl
nlv     ld a,19
        ld (6ØØØ5),a
        ret
        org 58Ø35
ti      *
        org 58281
msk     *
```

The origin here is obviously the address you call with your RANDOMIZE USR command to set the game going. Make a note of it.

The labels that lead instructions here are not called within this piece of programming.



They are called from other pieces of programming that will be published later. These are used to initialize the game when Willie has been killed and you have to start again.

THE SET-UP

The first thing the routine does is called the **ti** subroutine. This is the routine that prints the titles and the instruction pages call other routines given so far.

The various score parameters are then initialized. The score itself is stored in memory locations 57,331 to 57,342. The number of lives is stored in 57,343. And the level is stored in 57,344.

So 5 is stored in 57,343 via the one byte A register to start the player off with five lives. And Ø is stored in the level store 57,344, so that the game starts off on level one.

The score is set to zero by loading 57,337 and 57,338, 57,339 and 57,34Ø, and 57,341 and 57,342 with Ø, via the two byte HL register pair.

Then A is loaded with 19 and the music routine, **msk** is set to play 19 notes of Greensleeves.

Q+A

When do I start putting the game together?

Cliffhanger has been designed so that you can either enter and assemble the whole thing in one go at the end or build it up week by week. As far as possible the program does not call routines coming in future issues. But it may call data and routines in earlier parts, so make sure these are in memory when you execute a routine to test it.

If you have not assembled a routine that is called by the one you want to test, try putting a return instruction in at the point it is called. This will save you from a crash.

C=

The following routine sets up the game so that its levels and lives are set as they should be at the beginning of each game:

```
ORG  25856
LDA  #1
STA  49152
LDA  #5
STA  49153
LDA  #4Ø
STA  49164
STA  49154
LDA  #15
STA  5328Ø
STA  53281
LDA  #1Ø
STA  53287
LDA  #2
STA  53288
LDA  #1
STA  53289
LDA  #5
STA  5329Ø
STA  53291
STA  53292
LDA  #Ø
STA  54276
STA  54283
STA  5429Ø
LDA  #15
STA  54296
LDA  #67
STA  5ØØØØ
LDA  #4
STA  5ØØ1Ø
LDA  #112
STA  5ØØØ1
LDA  #4
STA  5ØØ11
LDA  #23Ø
STA  2Ø4Ø
LDA  #232
STA  2Ø41
LDA  #237
STA  2Ø42
LDA  #234
STA  2Ø43
STA  2Ø44
STA  2Ø45
LDA  #233
STA  2Ø46
RTS
```

Memory location FFD2 is in the Kernal ROM. This routine is the one that PRINTs a character string. The accumulator is used to carry parameters into that routine and loading A with 8, jumping to the subroutine at FFD2, then loading A with 142 and jumping to the subroutine has exactly the same effect as:

```
PRINT CHR$(8);CHR$(142)
```

This, you may remember from BASIC, disables the C= key. This is important. Hitting the C= key during the game—especially if it is shifted—can cause problems.

Next the sprite collision flags have to be cleared, otherwise when the game is reset it will start on level two. This is done by loading the accumulator with the contents of the two collision flags at 53,278 and 53,279 on the VIC chip.

ON THE LEVEL

The game starts off each time on level one, so 1 is loaded into the accumulator. This is then stored in the memory location which holds the level, 49,152.

Willie has five lives to start with, so 5 is loaded into the accumulator and stored in the 'lives' memory location, 49,153.

The sea is controlled by the variable in 49,164 and the delay which tells the game how fast to go is in 49,154. These are both initialized to 4Ø. All the memory locations mentioned above are part of the variable table you have set up in RAM.

COLOUR SET

Memory locations 53,28Ø and 53,281 are on the VIC chip. 53,28Ø controls the border colour and 53,281 controls background colour zero. These are set to 15 which puts the screen into multi-colour mode.

Memory locations 53,287 to 53,292 inclusive are on the VIC chip too and set the colours of the first six sprites.

THE SOUND OF SILENCE

The next thing to do is to cope with the sound output. Locations 54,276, 54,283 and 54,296 are all on the SID chip. And they're not really memory locations at all, they're control registers.

54,276 controls voice one, 54,283 controls voice two, and 54,296 is the register that selects the filter mode and volume. Saying that they are registers means that each bit controls a different input/output function. For example, bit seven of the voice control

registers selects a random noise waveform. Bit six selects a pulse waveform, bit five a sawtooth waveform and bit four a triangular waveform. Bit three is a test bit which switches the oscillator off. Bit two controls ring modulation, bit one the synchronization and bit zero is the gate which either starts the note off or tells it when to begin cutting off the note.

The filter mode and volume register acts similarly bit by bit. Bit seven controls the cut-off of voice three. Bits six, five and four select the high-pass, band-pass and low-pass filters respectively. And bits three to zero select the output volume which can be set anywhere between Ø and 15.

Each of the register's various functions are controlled by setting, or resetting, the appropriate bit. Generally, setting a bit to 1 switches on a particular function, and resetting it to Ø switches that function off again.

At the beginning of the game, you want to switch all these functions off to clear out any bits left over from sounds made before. So Ø is loaded into the accumulator and stored in 54,276, 54,283 and 54,29Ø.

GULLS AND SPRITES

At the beginning of the game the seagulls have to be moved back to their start positions. The screen coordinates of the gulls change as the game goes on, so their values are stored in the variable table in RAM too, at memory locations 5Ø,ØØØ and 5Ø,ØØ1, 5Ø,Ø1Ø and 5Ø,Ø11. And the start coordinates of the gulls are stored in those locations.

Memory locations 2,Ø4Ø to 2,Ø47 are in ROM. They are the sprite data pointers. The numbers in there tell the processor where to find the data for the sprites. You'll notice that the first seven sprite data pointers are loaded, while only the first six had their colours set. In fact, seven sprites are being used, but one of them is empty. And, although it needs null data to make it empty, it does not need to have a colour set as it only uses the background colour.

LIVES AND LEVELS

At the beginning of each screen, you have to set the number of levels and lives to the right value. The following routine does just that:

```
ORG   25393
LDX   49153
LDY   #Ø
LDA   #27
LOOPI STA 1Ø7Ø,Y
INY
DEX
BNE   LOOPI
LDA   LEVEL
CMP   #6
BNE   C
LDA   #1
STA   49152
INC   49153
DEC   49154
C     LDX 49152
LDY   #Ø
LDA   #9
LOOP  STA 111Ø,Y
INY
DEX
BNE   LOOPJ
RTS
```

The number of lives Willie has left are stored in 49,153. And the contents of this location are loaded into the X register. The index register Y is then set to Ø.

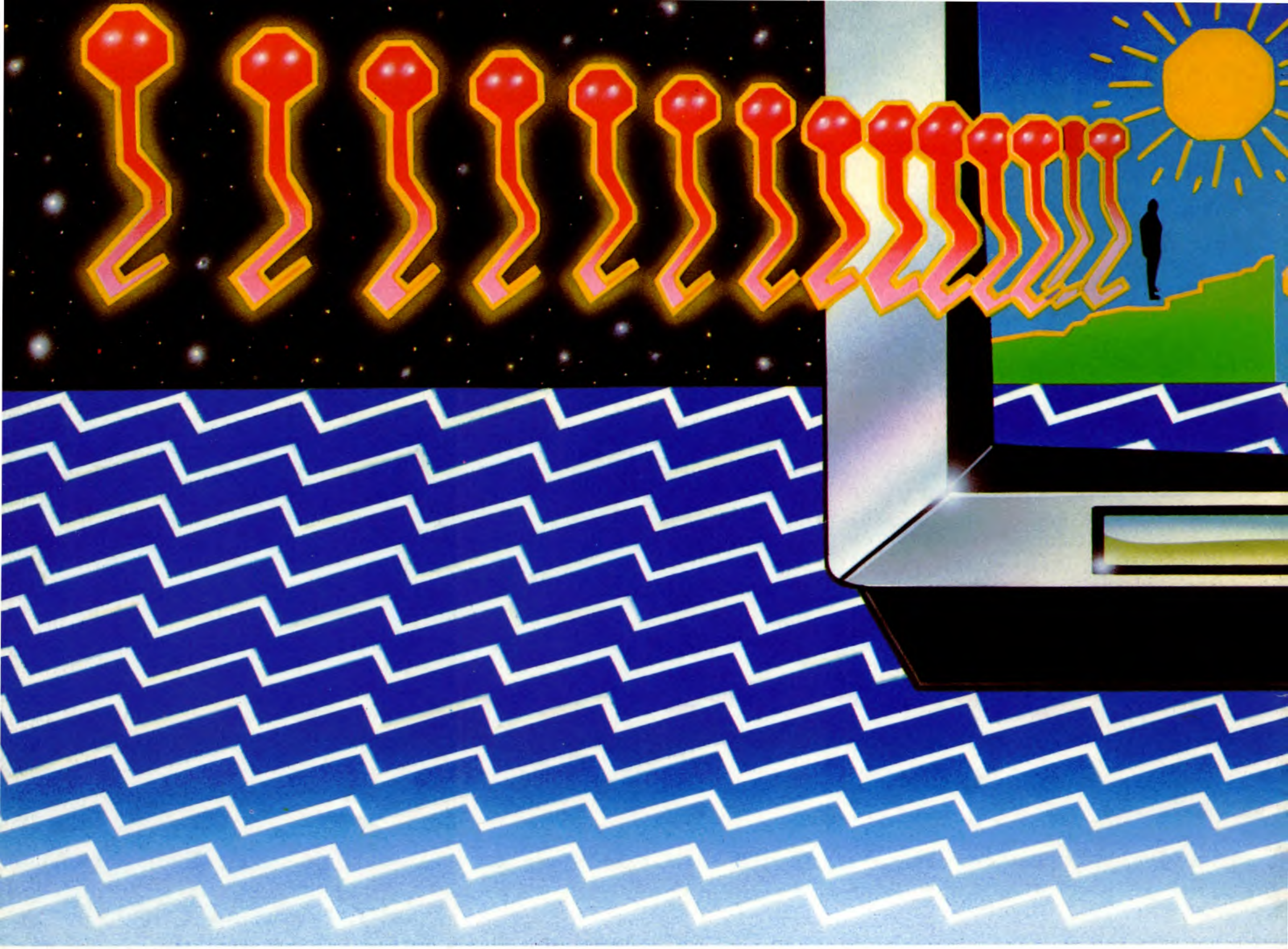
The accumulator is then loaded with 27, which is the UDG number of the man graphic. This is stored on the screen at location 1,Ø7Ø offset by Y. Y is incremented and X decremented. And if X has been decremented to zero, the processor loops back.

So the processor goes round the loop the same number of times as the lives Willie has left. Each time it prints up a little man graphic on the next position on the screen. This gives as many men as lives left.

The number of the level you are on is stored in 49,152. This is compared to six. If it's not six the processor branches forward over the next little routine.

Level five is the top level in the game. So if the value in the level variable stored at 49,152 has reached six, it is set back to one by loading 1 into the accumulator and storing it in 49,152.

For completing level five—and nominally reaching level six—you earn an extra life. So the life variable in 49,153 is incremented. But because you are so clever, the advance of the sea is speeded up, so the delay variable in 49,154 is decremented. Next comes the loop which prints up the number of the level you're on, on the screen. It works exactly the same way as the lives loop above. This time, though, the contents of the level variable are loaded into the loop counter X, a different screen position is used and the



number 9 is loaded into the accumulator and stored in the screen. This is the screen code for a capital I (or a 1 in Roman numerals).

The Acorn's initialization routine calls many of the routines given so far, and a few that haven't, such as the ones at 1A2E and 1A3C. So don't call it until there are programs at all the addresses it jumps to, which will be given in later issues. Type PAGE=&3000, NEW (and *TAPE; if you have a DFS) as normal to type it in and assemble it.

```
70 DATA1
80 DATA4
90 DATA2
100 DATA5
110 DATA6
120 FORA%=&1B2DTO&1B31:READ?A%:NEXT
160 FORPASS=ØTO3STEP3
170 P%=&1B32
180 [OPTPASS
190 .PtSc
200 LDA#22
210 JSR&FFEE
220 LDA#2
230 JSR&FFEE
240 JSR&1483
250 JSR&1855
260 JSR&182D
270 JSR&1A2E
280 JSR&1A3C
290 JSR&1894
300 LDX&83
310 LDA&1B2D,X
320 AND#&1
330 BEQLb1
340 JSR&193D
350 .Lb1
360 LDX&83
370 LDA&1B2D,X
380 AND#&2
390 BEQLb2
400 JSR&19B9
410 .Lb2
420 JSR&1AFF
430 RTS
440 ]NEXT
```

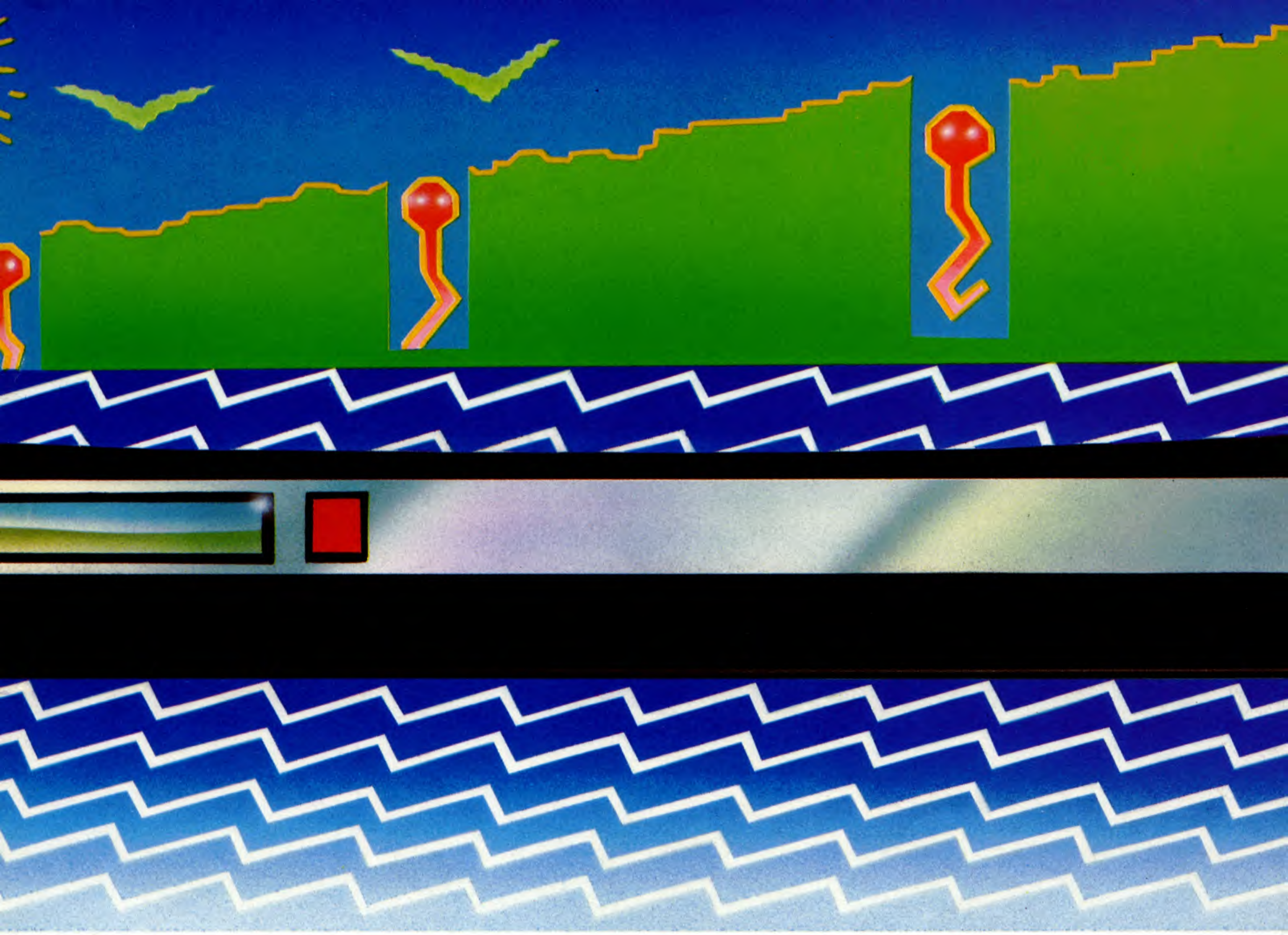
SCREEN BY SCREEN

The DATA tells the program what elements appear on which screen. The 1 specifies potholes, the 4 boulders and the 2 snakes. So 5 specifies potholes and boulders and 6 specifies snakes and boulders. This is the order that the obstacles come in. Again, this DATA is read into a data table—from &1B2D to &1B31—where the machine code program can access it.

The first thing the assembly language program does is change MODE. Loading 22 into the accumulator and jumping to the routine at &FFEE, does that. The 2 loaded up after that and output through the &FFEE routine makes the change to MODE 2. Then the processor jumps to a series of subroutines —in Lines 24Ø to 29Ø—which set the game going.

The routine at &1483 switches the cursor off. The one at &1855 sets the colours. The one at &182D defines the characters. The one at &1A2E prints the headings, seagulls and clouds. The one at &1A3C prints the score and lives. And the one at &1894 prints up the slope.

The memory location at &83 is used to store the number of the level the player has reached. So it also specifies which screen is required. The level number is loaded from



&83 into the X register so that it can be used as an index.

The accumulator is loaded with data from the screen data table, offset by the level number in X. And ANDing it with 1 asks where bit zero is set and potholes are required.

If this bit is not set, the BEQ jumps the next instruction. If it is set, the BEQ's condition is not fulfilled and the processor jumps to the subroutine at &1B2D, which prints the potholes over the contour of the slope.

The appropriate data byte is loaded up again and ANDed with 2 this time. This checks whether bit one is set and whether snakes are required.

Again, if this bit is not set, the BEQ jumps the next instruction. If it is set, the BEQ's condition is not fulfilled and the processor jumps to the subroutine at &19B9, which prints the snakes over the contour of the slope. The printing up the boulders are dealt with elsewhere.

Finally in this section, processor jumps to the subroutine at &1AFF. This is the one that prints up Willie's goodies at the top of the slope.

This is the routine that fires the Dragon version of Cliffhanger into action.

```
         ORG    19426
GBIN     JSR    START
         LDA    #5
         STA    18239
         CLR    18238
         LDX    #1824Ø
         LDB    #6
GBINI    CLR    ,X+
         DECB
         BNE    GBINI
         RTS
START    EQU    19ØØØ
```

The origin here is obviously the address you call with your EXEC command to set the game going. Make a note of it.

The labels that lead instructions here are not called within this piece of programming. They are called from other pieces of programming that will be published later. These are used to initialize the game again when Willie has been killed and you have to start all over again.

MAKING A START

The first thing this routine does is to jump to the subroutine labelled START. This is the routine that prints the title and instruction pages and starts the game going. The next thing that has to be done is to initialize all the score parameters.

The level the player has reached is stored in memory location 18,238. The number of lives is stored in 18,239. And the score itself is stored in the six bytes from 18,24Ø onwards.

So 5, the number of lives Willie starts out with, is loaded into the accumulator and stored in 18,239. Memory location 18,238 is then cleared to set the level to zero—which gives level one.

The X register is then loaded with the location of the first score byte and B is loaded with 6—the number of score bytes. CLR ,X+ then clears the byte pointed to by X and increments X. DECB decrements B and the BNE instruction branches back to clear the next byte if the B has counted down to Ø. This little routine sets all the memory locations 18,24Ø to 18,245—that is, those set aside to store the score—to Ø, one after another.

HOW BASIC PROGRAMS ARE STORED

The computer's inner workings may seem somewhat obscure, especially when things start to go wrong. But if you understand how a program is stored you're well on the way to sorting out the bugs

Have you ever wondered what the computer does with a program when you type it in? You probably know that it stores it in a special area of memory reserved for BASIC programs, but do you know where this is, and do you know what the program looks like when it is converted into something the computer can understand?

In fact the program in the computer's memory is very similar to the characters you type in and a lot of the numbers in memory are ASCII codes of letters from the program. But the computer shortens some words—the keywords—and puts in extra information, literally between the lines, to help it keep track of where it is.

The computer also has a separate area where it stores the variables set up by the program as these change while the program is running and need to be updated continuously. So when you type 1Ø LET A=6, say, the computer stores the line as part of your program and also puts the name and value of the variable into the variable store.

Now to start exploring. First enter the following simple program. Type it in exactly as shown. On the Spectrum don't enter any spaces at all. On the other computers enter a space *only* after the words PRINT and LET. You can now use this to discover how the computer stores programs.

```
1Ø PRINT "HELLO"
2Ø LET A=2
3Ø PRINT A
4Ø STOP
```

There are two memory addresses in the Spectrum which together act as a pointer to the start of the BASIC program—these addresses are 23635 and 23636. To find out where your BASIC program starts, enter the following direct command:

```
PRINT PEEK 23635+256*PEEK 23636
```

This will normally give the answer 23755 for a 48K Spectrum. If you find that your computer gives a different answer and you have typed the command in correctly, then change 23755 in the following command to your value. Now type in this direct command

```
FOR I=23755 TO 23755+4Ø:PRINT PEEK
   I;"□";:NEXT I
```

You should get the following output. If not, check that you have typed the command correctly:

```
Ø    1Ø   9    Ø    245  34   72   69
76   76   79   34   13   Ø    2Ø   11
Ø    241  65   61   5Ø   14   Ø    Ø
2    Ø    Ø    13   Ø    3Ø   3    Ø
245  65   13   Ø    4Ø   2    Ø    226
13
```

Now, using the ASCII codes for letters and numbers, see if you can make sense of this:

```
?    ?    ?    ?    ?    "    H    E
L    L    O    "    ?    ?    ?    ?
?    ?    A    =    2    ?    ?    ?
?    ?    ?    ?    ?    ?    ?    ?
?    A    ?    ?    ?    ?    ?    ?
?
```

You can see part of the BASIC program emerging already, but there are still a fair number of question marks.

You might expect PRINT to precede the "HELLO" but instead the list shows the value 245. The Spectrum uses values above 164 to signify BASIC keywords. If you consult a list of these in the manual you'll find that 245 is the value, called the *token*, used for PRINT. Looking at the values for the tokens, you can now fill in the keywords:

```
?    ?    ?    ?    PRINT"    H    E
L    L    O    "    ?    ?    ?    ?
?    LET  A    =    2    ?    ?    ?
?    ?    ?    ?    ?    ?    ?    ?
PRINTA    ?    ?    ?    ?    ?    STOP
?
```

If you look at the 2nd, 15th, 30th and 37th bytes in the original table you will notice a pattern ... 1Ø, 2Ø, 3Ø, 4Ø. These bytes are used to hold the low order part of the line number, the preceding byte is used for the higher order part. The next two bytes in the line, low order byte first this time, are used to store the length of the line, excluding these first four bytes.

You now have the following picture:

```
Ø    1Ø   lenL lenH PRINT"    H    E
L    L    O    "    ?    Ø    2Ø   lenL
lenH LET  A    =    2    ?    ?    ?
?    ?    ?    ?    Ø    3Ø   lenL lenH
PRINTA    ?    Ø    4Ø   lenL lenH STOP
?
```

Now for the final few bytes. The value 13 is used to indicate end-of-line (eol). The value 14 indicates that the following four bytes are the coded form for a numerical constant, in this case 2. The code 14 is used to indicate a numerical constant in the variable list table (vlt). Here, then, is the final picture:

```
Ø    1Ø   lenL lenH PRINT"    H    E
L    L    O    "    eol  Ø    2Ø   lenL
lenH LET  A    =    2    vlt  Ø    Ø
2    Ø    Ø    eol  Ø    3Ø   lenL lenH
PRINTA    eol  Ø    4Ø   lenL lenH STOP
eol
```

The Spectrum places a number greater than 4Ø in the higher order byte of the line number field to signify end of program.

The command NEW erases any old programs and variables in the computer—you cannot retrieve it, so beware.

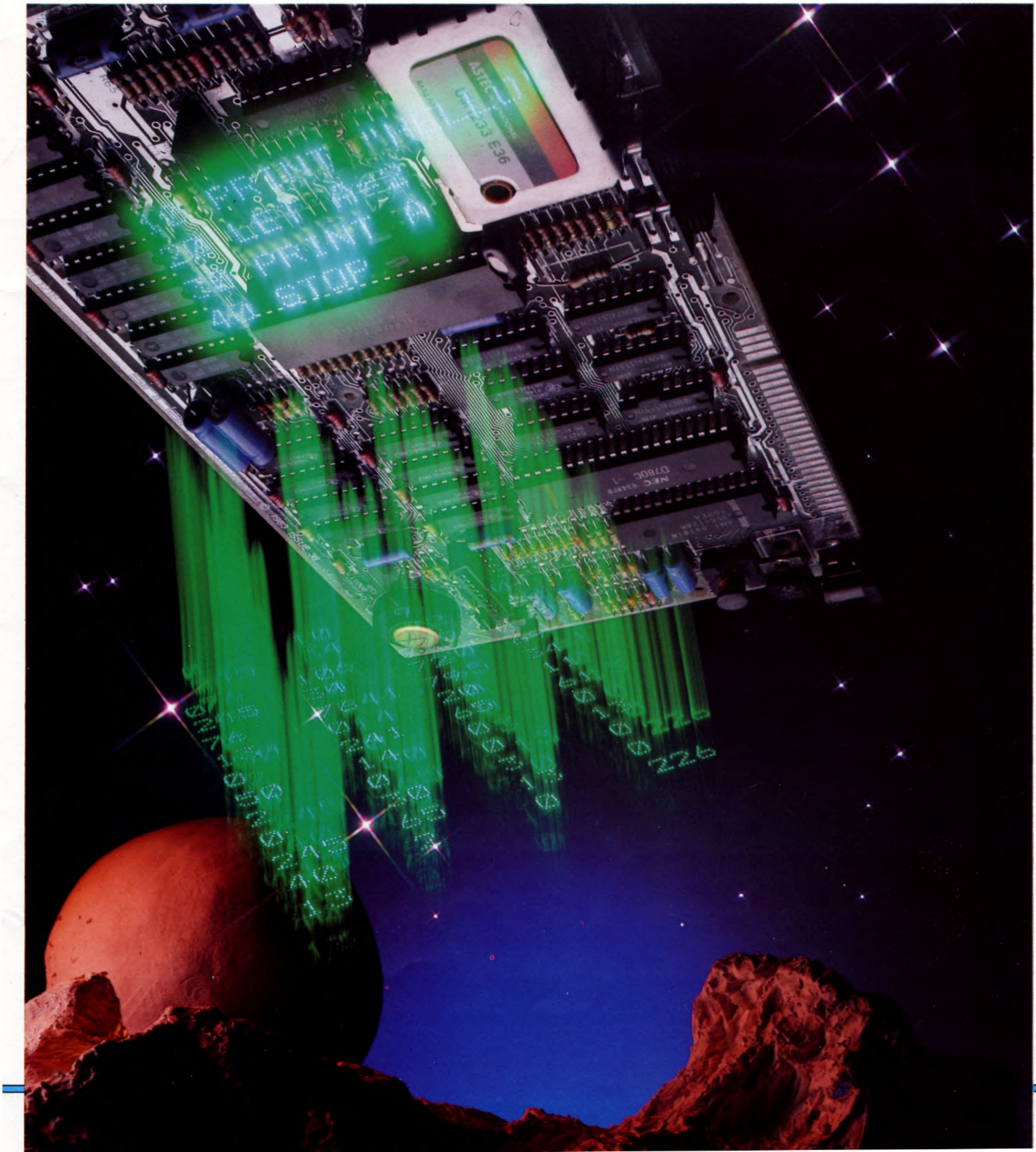
STORING THE VARIABLES

The last program showed how a BASIC program was stored in memory, but any variables set up by the program are stored separately in a special variable store. There are two memory addresses—23627 and 23628—that contain the address of the start of the store, and another two—23641 and 23642—that point to the end.

Try out this program to print the entire contents of the store:

```
1Ø FOR A=PEEK 23627+256*PEEK 23628
   TO PEEK 23641+256*PEEK 23642
2Ø PRINT PEEK A;"□□";
3Ø NEXT A
```

- FINDING THE START OF BASIC
- PEEKING THE MEMORY
- TRANSLATING THE NUMBERS
- ASCII CODES AND TOKENS
- HOW THE PROGRAM IS STORED
- POINTING TO THE VARIABLES STORE
- NUMERIC VARIABLES
- STRINGS AND ARRAYS
- HOW THE MEMORY IS USED



NUMERIC VARIABLES

First set up a variable:

```
1 LET B=15ØØØØ
```

RUNning the program gives the following output:

```
98 146 18 124 Ø Ø 225 Ø
 B     ← value →
```

The 98 is ASCII code for B plus 32, this shows the computer it is dealing with a numeric variable. The 15ØØØØ is stored in the next five bytes in floating point format (see page 79Ø). The last two bytes 225 and Ø signify the end of the variables store. They are always there and so will be missed out from the next examples.

Long variable names are stored in a similar way although their names are scrambled rather more. Try this:

```
1 LET DAVID=3Ø
```

This gives:

```
164  97  118  1Ø5  228  Ø  Ø  3Ø  Ø  Ø
 D    A    V    I    D   ←    value   →
```

The first five bytes store the name DAVID except that the first letter has 96 added to it to indicate it is a long name, the middle ones have the usual 32 and the last letter has 16Ø added on to indicate the end of the name. The last five bytes store the number.

STRINGS AND ARRAYS

The next program line shows how strings are stored:

```
1 LET A$="STRING"
65   6   Ø   83   84   82   73   78   71
A            S    T    R    I    N    G
```

The name of the string, A, comes first followed by two bytes giving the length of the text. ASCII codes for the string itself follow immediately afterwards.

Now try a numeric array:

```
1 DIM F(2)
2 LET F(1)=1ØØ
3 LET F(2)=2ØØ
134  13  Ø  1  2  Ø  Ø  Ø  1ØØ  Ø  Ø
 F                   ←   value      →
Ø  Ø  2ØØ  Ø  Ø
←    value    →
```

The first byte is F plus 64 for a numeric array variable. The next two contain the number of bytes following, low byte first. Count them, you'll find there are 13. The next byte gives the number of dimensions and the following two store the number of elements reserved. After that are the two numbers, each stored in five bytes.

Finally, try a string array:

```
1 DIM B$(2,6)
2 LET B$(1)="STRING"
3 LET B$(2)="ARRAY"
194 17 Ø 2 2 Ø 6 Ø
 B
83 84 82 73 78 71 65 82 82 65 89 32
S  T  R  I  N  G  A  R  R  A  Y
```

The first byte is the ASCII code of B plus 128 for string arrays. The next two give the number of bytes following. The next byte is the number of dimensions—in this case 2, followed by two pairs giving the number of elements in the array and the maximum length of each element. These are immediately followed by the strings themselves. The space at the end is added on to the word ARRAY to full up the six characters allocated to each string.

There are two memory addresses in the 64 which together act as a pointer to the start of the BASIC program. These addresses are 43 and 44. To find out where your BASIC program starts enter the following:

```
PRINT PEEK (43)+256*PEEK (44)
```

This will normally give you the answer 2Ø49. If you find that your computer gives a different answer and you have typed the command in correctly, then change 2Ø49 in the following command to your value:

```
FOR I=2Ø49-1 TO 2Ø49+39:PRINT PEEK
   (I);:NEXT I
```

You should get the following output, if not check that you have typed the command correctly:

```
Ø    15   8    1Ø   Ø    153  32   34
72   69   76   76   79   34   Ø    25
8    2Ø   Ø    136  32   65   178  5Ø
Ø    33   8    3Ø   Ø    153  32   65
Ø    39   8    4Ø   Ø    144  Ø    Ø
Ø
```

Now using the ASCII codes for letters and numbers, see if you can make some sense of this:

```
?   ?   ?   ?   ?   ?   □   "
H   E   L   L   O   "   ?   ?
?   ?   ?   ?   □   A   ?   2
?   ?   ?   ?   ?   ?   □   A
?   ?   ?   ?   ?   ?   ?   ?
?
```

You can see part of the BASIC program already but there are still a fair number of question marks. Note that the space box—□—represents a space.

You might expect to find the word PRINT preceding "HELLO" but instead there is the value 153. The computer uses values above 127 to indicate BASIC keywords. If you consult a list of these keywords in the manual you'll find that 153 is the value, called the *token*, used for PRINT. You can now fill in the keywords:

```
?    ?    ?    ?    ?    PRINT□   "
H    E    L    L    O    "    ?   ?
?    ?    ?    LET  □    A    =   2
?    ?    ?    ?    ?    PRINT□   A
?    ?    ?    ?    ?    STOP ?   ?
?
```

If you look at the 3rd, 17th, 27th and 35th bytes you will notice a pattern . . . 1Ø, 2Ø, 3Ø, 4Ø. These bytes are used to hold the low order byte of the line number, the following bytes are used to store the high order byte. The two bytes before the line numbers, again low byte first, are pointers to the start of the next line. For example, the first two bytes, 15 and 8, point to 15+256*8 which equals 2Ø63. Since the address of the first byte is 2Ø48 you can easily count along to check that the address 2Ø63 is indeed the start of the next set of pointers. This is shown more clearly below. Note that the computer uses a Ø to mark the end-of-line (eol):

```
Ø    ptrL ptrH 1Ø   Ø    PRINT□   "
H    E    L    L    O    "    eol prtl
ptrH 2Ø   Ø    LET  □    A    =   2
eol  ptrL ptrH 3Ø   Ø    PRINT□   A
eol  ptrL ptrH 4Ø   Ø    STOP eol Ø
Ø
```

The computer signifies the end of a program by setting the pointer fields (ptrL and ptrH) equal to zero, so a BASIC program always ends with three zeros.

When you type NEW, the first two pointer bytes are set to zero so it is fairly easy to reset these. The high pointer is always 8 and the low pointer should equal the length of the first line counting from the zero to the eol marker.

STORING THE VARIABLES

The Commodore has three areas where it can store variables. Simple variables are stored immediately after the BASIC program and locations 45 and 46 hold the address of the start of the store. Array variables are stored next, with the start address kept in locations 47 and 48. And two more locations—49 and 5Ø point to the end of the variable store. Any strings are stored at the other end of the RAM. The start address is given by the contents of locations 51 and 52 and the end address by locations 53 and 54.

SIMPLE VARIABLES

All these locations allow you to find out exactly where the variables are, and the next program actually looks through the simple variable store to see what happens to the variable A. The numbers underneath are the contents of the store and if any of these are ASCII codes they have been translated for you:

```
1 AB=123
1Ø FOR Z=PEEK(45)+PEEK(46)*256 TO
   PEEK(47)+PEEK(48)*256-1
2Ø PRINT PEEK(Z);:NEXT Z
```

```
65   66   135   118   Ø   Ø   Ø
A    B    ←       value       →
9Ø   Ø    14Ø   5    48   Ø   Ø
Z    ←          value         →
```

The last seven bytes store the variable Z and constitute the part of the program used to look at the memory. So to save confusion these bytes will be left out from now on. The variable you're interested in is AB and it is simply stored as the ASCII codes for A and B followed by the value of the variable in floating point format.

If you change Line 1 to a string variable you'll get the following numbers:

```
1 ZZ$="TEXT"
```

```
9Ø 218 4 1Ø 8 Ø Ø
Z  Z
```

The ASCII codes for the variable's name ZZ comes first, but the second number has 128 added on to it to indicate a string. The next number gives the number of characters in the string and the next two bytes are the low and high pointers to the position of the string in memory. In this case it is 1Ø+8*256 which equals 2Ø58. This is in fact the position of the word TEXT in the BASIC program. However, if the variable changed in the program or it was originally undefined (as in an INPUT), then the pointers would point to the string store which is at the top end of RAM.

ARRAY VARIABLES

A string array is stored in a similar way to a string. Try the next program and see. Notice that the pointers in Line 1Ø are now pointing to the start and end of the array store:

```
1 DIM AA$(2)
2 AA$(1)="APPLE"
3 AA$(2)="PLUM"
1Ø FOR Z=PEEK(47)+PEEK(48)*256 TO
   PEEK(49)+PEEK(5Ø)*256-1
2Ø PRINT PEEK(Z);:NEXT Z
```

```
65 193 16 Ø 1 Ø 3 Ø Ø Ø 5 26 8 4 45 8
```

As usual this starts with the name and again 128 is added on the second letter because this is a string variable. The next number, 16, is the number of bytes taken up in the store. Count them and check that there are 16. The next pair gives the number of dimensions and the pair after that the number of elements reserved by the DIM statement. Following these numbers are three bytes per element. As the first element AA$(Ø) is not defined, the first three are zero. For the others, the first byte gives the number of characters in the string and the next two point to the position of the string.

Finally, try a numeric array, a two-dimensional one this time:

```
1 DIM AA(Ø,2)
2 AA(Ø,Ø)=1
3 AA(Ø,2)=2
```

```
65  65  24  Ø  2  Ø  3  Ø  1  129 Ø  Ø  Ø  Ø
                               ←  value  →
Ø    Ø    Ø    Ø    Ø    13Ø   Ø    Ø    Ø    Ø
←       value        →   ←        value        →
```

You should be able to work out the first five bytes by now. The next two pairs hold the number of elements in each dimension but in the opposite order to the DIM statement. These are then followed by the values of the variables in floating point format.

The Acorn computers have a BASIC variable called PAGE which contains the address of the first byte of the program. PAGE can be changed, but it is &EØØ when you first switch on unless you have a disk drive. Now type in this direct command

```
MODE1: PRINTTAB(Ø,1Ø): FOR I=PAGE TO
   PAGE+36:PRINT ?I;:NEXT I
```

You should get the following output, if not check that you have typed the command in correctly.

```
13   Ø    1Ø   13   241  32   34   72
69   76   76   79   34   13   Ø    2Ø
9    233  32   65   61   5Ø   13   Ø
3Ø   7    241  32   65   13   Ø    4Ø
5    25Ø  13   255  Ø
```

Now using the ASCII codes for letters and numbers, see if you can make some sense of this:

```
?    ?    ?    ?    ?    □    "    H
E    L    L    O    "    ?    ?    ?
?    ?    □    A    =    2    ?    ?
?    ?    ?    □    A    ?    ?    ?
?    ?    ?    ?    ?
```

You can see part of the BASIC program already but there are still a fair number of question marks which represent out of range codes. (Note the space box—□—is used to represent a space.)

You might expect to find PRINT preceding "HELLO" but instead there is the value 241. The Acorns use values above 127 to signify BASIC keywords. If you consult a list of these keywords in the BASIC manual you'll find that 241 is the value, called the *token*, used for PRINT. You can now fill in the keywords:

```
?    ?    ?       ?        PRINT□   "    H
E    L    L       O        "    ?    ?    ?
?    LET  □       A        =    2    ?    ?
?    ?    PRINT□  A        ?    ?    ?
?    STOP ?       ?        ?
```

You now have to discover what the four bytes at the start of the line are used for. If you look at the 3rd, 16th, 25th and 32nd bytes you will notice a pattern—1Ø, 2Ø, 3Ø, 4Ø. From this you can deduce that these bytes are used to hold the line number. The first byte of each line, ASCII 13, is used to indicate the start of the line (sol). The 4th byte is used to contain the length of the line (len), this count includes the bytes used for the line number as well as the first byte.

You now have the following picture:

```
sol  Ø     1Ø    len     PRINT□   "    H
E    L     L     O       "    sol  Ø    2Ø
len  LET   □     A       =    2    sol  Ø
3Ø   len   PRINT□  A     sol  Ø    4Ø
len  STOP  sol   255     Ø
```

After the final sol there is the value 255 in the higher order byte of the line number. If you multiply 255*256 you will get an answer which exceeds the allowable line number on the BBC. This is how the BBC signifies end of program.

All BASIC programs end with 255 and Ø (or &FF and Ø in hex), and the variable TOP always points to the next byte after Ø. In fact TOP is always two bytes more than PAGE because these two bytes are present.

Typing NEW on the BBC sets the second byte of the program, location PAGE+1, to 255. Typing OLD will change it back to Ø. If you want to see this for yourself then enter the following direct command:

```
P. ?PAGE, ?(PAGE+1), ?(PAGE+2),
   ?(PAGE+3), ?(PAGE+4)
```

Then type NEW followed by the same direct command and you will notice that location PAGE+1 has changed its value from Ø to 255. The computer sees this in the high order byte of the line number, so assumes this to be the end of the program. The program is not lost, it's just that the computer doesn't know it is there. Typing OLD resets the second byte to zero.

STORING THE VARIABLES

The Acorn computers use a special variable called LOMEM which contains the address of the first byte of the variable store. LOMEM is normally the same as TOP so the variables sit immediately above a program, but it can be changed. The address of the top of the store is contained in memory locations 2 and 3; there's no special variable name in this case.

The following one-line program prints out the entire variable store: Type PAGE = &ØØ*TAPE and NEW before typing this in. Don't type any spaces except those indicated.

```
1Ø FOR A=LOMEM□ TO ?2+256*?3-1:
   PRINT;?A;"□";:NEXT:PRINT
```

If you type RUN you'll see the following:

```
Ø Ø 14Ø 98 16Ø Ø Ø
```

These store the last value of the variable A used in the program above, and to save confusion they will be missed off all the following examples.

NUMERIC VARIABLES

Now try setting up a variable. Enter this line and type RUN, the numbers show the output, and the ASCII codes have been translated into letters underneath:

```
1 HELLO=123
Ø  Ø  69  76  79  Ø  135  118  Ø  Ø  Ø
      E   L   L   O  ←    value       →
```

The first two bytes are pointers which hold the address of the next variable beginning with the same initial letter—there isn't one in this case so the bytes are zero. If you get some other number in the first byte don't worry. This is the least significant byte and is not cleared by the computer so it takes on whatever value was in that location before. If the second byte (the most significant byte) is zero then the other is assumed to be zero as well. The pointers are followed by the ASCII codes for the variable name minus the first letter, then a zero byte then the value of the variable in floating point format.

Now try an integer variable:

```
1 NUMBER%=123
Ø  Ø  85  77  66  69  82  37  Ø
      U   M   B   E   R   %
123 Ø Ø Ø
← value →
```

Again, there are two pointer bytes followed by the ASCII for UMBER% and a zero. But this time the number is stored in integer format, taking only four bytes. Using integers saves memory and helps to speed up a program.

STRING VARIABLES

String variables are slightly more complicated, try this:

```
1S$="A STRING"
Ø   Ø   36  Ø   62  14  16  8
        $
65  32  83  84  82  73  78  71
A       S   T   R   I   N   G
```

As usual the first two bytes are a pointer to the next variable starting with S. Then there's the variable name minus the first letter, which just leaves the $ sign for a single letter variable like this. Then there's the usual zero byte. The next two numbers—62 and 14—hold the address of where the string is stored. The next number gives the total number of bytes taken up in the store (count them, there are 16). The next number—8—gives the number of bytes in the string, and this is followed by the ASCII codes for A STRING.

Have a look at the numbers 62 and 14 again. These point to address 62+14*256 which equals 3646. This gives the address of the first letter in the string—the A. To prove to yourself that this is correct, type PRINT LOMEM to find the start of the store and add on the eight bytes to bring you to the A in the numbers above. The two numbers should be the same.

ARRAY VARIABLES

Try a string array first:

```
1 DIMWORD$(1)
2 WORD$(Ø)="ZERO"
3 WORD$(1)="ONE"

Ø  Ø  79 82 68 36 4Ø Ø  3  2  Ø  1Ø6  14
      O  R  D  $  (
4  4  11Ø   14 3  3  9Ø 69 82 79 79 78 69
                     Z  E  R  O  O  N  E
```

This starts off similar to a string variable—the pointer followed by ASCII for ORD$(and a zero. The next number tells you how many dimensions there are, worked out as 2*dims+1. So a one-dimensional array like this has the number 3, a two-dimensional array has 5 and so on. The next two bytes store the number of elements set up by the DIM statement. Each element now has four bytes reserved for it; the first two bytes—1Ø6 and 14—give the address of the start of the word ZERO and the next byte (repeated) gives the number of characters in the string. After these come the strings themselves.

Now try a numerical array:

```
1 DIMNUM(1,1)
2 NUM(1,1)=4Ø
3 NUM(1,Ø)=2Ø
4 NUM(Ø,Ø)=1Ø

Ø    Ø   85  77  4Ø  Ø   5   2   Ø   2   Ø
         U   M   (
132  32  Ø   Ø   Ø   Ø   Ø   Ø   Ø   Ø
←    value       →   ←   value       →
133  32  Ø   Ø   Ø   134  32  Ø   Ø   Ø
←    value       →   ←    value       →
```

This starts off fairly normally. The number 5 shows there are two dimensions since 2*2+1 equals 5. The next two pairs of bytes give the number of elements in each dimension—this is a 2 by 2 array.

After this comes the elements themselves in the correct order, it doesn't matter in what order they were specified. The numbers 1Ø, 2Ø and 4Ø are stored as 132 32 Ø Ø Ø, 133 32 Ø Ø Ø and 134 32 Ø Ø Ø in floating point format. Notice that memory is reserved for element (Ø,1) and is set to Ø.

There are two memory addresses in the Dragon which together act as a pointer to the start of the BASIC program. These addresses are decimal 25 (hex 19) and decimal 26 (hex 1A). To find out where your BASIC program starts enter the following direct command:

```
PRINT PEEK(25)*256+PEEK(26)
```

If the computer has just been switched on the answer will be 7681. If you find that your machine gives a different answer and you have typed the command in correctly then change the 7681 in the following command to your value:

```
FOR I=7681 TO 7681+39:PRINT PEEK(I);:
   NEXT I
```

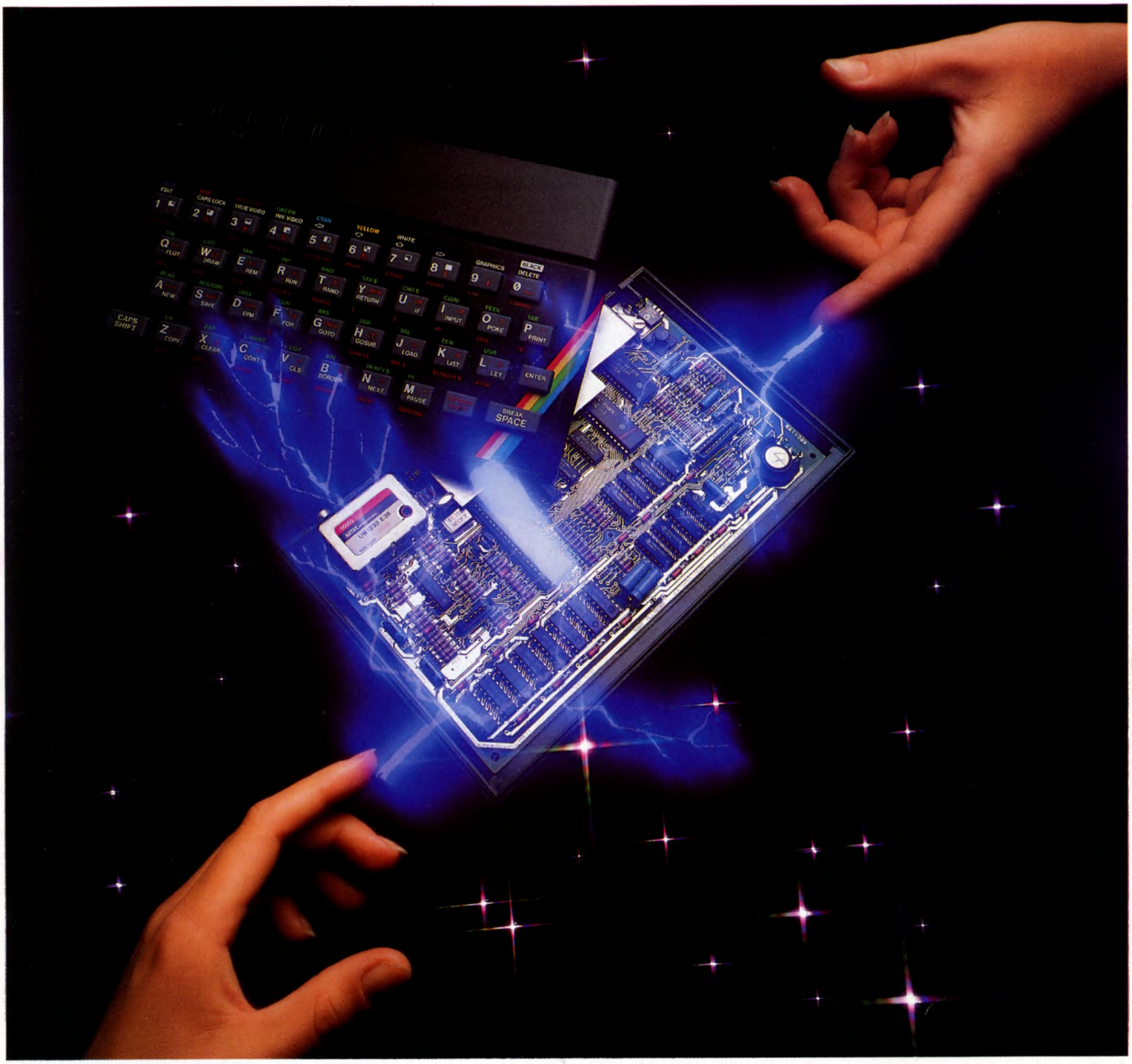
You should get the following output, if not, check that you have typed the command in correctly:

```
3Ø  15  Ø    1Ø  135  32   34  72
69  76  76   79  34   Ø    3Ø  25
Ø   2Ø  142  32  65   2Ø3  5Ø  Ø
3Ø  33  Ø    3Ø  135  32   65  Ø
3Ø  39  Ø    4Ø  146  Ø    Ø   Ø
```

Now, using the ASCII codes, for letters and numbers, see if you can make sense of this:

```
?   ?   ?   ?   ?   □   "   H
E   L   L   O   "   ?   ?   ?
?   ?   ?   □   A   ?   2   ?
?   ?   ?   ?   ?   □   A   ?
?   ?   ?   ?   ?   ?   ?   ?
```

You can see part of the BASIC program already, but there are still a fair number of



question marks. Note that the space box—□—means a space.

Looking at the above you might expect the word PRINT to precede "HELLO", but instead the value 135 appears. The computer uses values above 127 to signify BASIC keywords. The number 135 is the value, called the *token*, used for PRINT. With these values, you can now fill in the keywords:

```
?   ?   ?    ?   PRINT□   "   H
E   L   L    O   "    ?   ?   ?
?   ?   LET  □   A    =   2   ?
?   ?   ?    ?   PRINT□   A   ?
?   ?   ?    ?   STOP ?   ?   ?
```

Unfortunately, the manual doesn't list the values of the tokens but you can find out what they are using the following program:

```
1Ø CLEAR 1ØØØ:CLS
2Ø C=32819:FOR K=Ø TO 78
3Ø W$(K/39)=W$(K/39)+CHR$
   (PEEK(C))
4Ø C=C+1: IF PEEK(C-1)<128 THEN 3Ø
5Ø NEXT
6Ø C=33226: FOR K=1 TO 34
7Ø F$=F$+CHR$(PEEK(C))
8Ø C=C+1:IF PEEK(C-1)<128 THEN 7Ø
9Ø NEXT
1ØØ PRINT:INPUT"□INPUT TOKEN IN
   HEX□";T$
11Ø TK=VAL("&H"+T$)
12Ø IF TK>6528Ø THEN TK=TK-
   6528Ø:GOTO 21Ø
13Ø IF TK<128 OR TK>2Ø5 THEN 25Ø
14Ø K=-39*(TK>166):P=1
15Ø IF K=TK-128 THEN W$=W$
   (K/39):GOTO 18Ø
16Ø P=P+1:IF ASC(MID$(W$(K/39),
   P-1,1))<128 THEN 16Ø
17Ø K=K+1:GOTO 15Ø
18Ø PRINT:PRINT"□TOKEN□";
   T$;"□=□";
19Ø A$=MID$(W$,P,1):IF A$<CHR$
```

```
   (128) THEN PRINTA$;:P=P+1:
   GOTO 19Ø
2ØØ PRINTCHR$(ASC(A$)AND127):
   GOTO 1ØØ
21Ø K=Ø:P=1:IF TK<128 OR TK>161
   THEN 25Ø
22Ø IF K=TK-128 THEN W$=F$:
   GOTO 18Ø
23Ø P=P+1:IF ASC(MID$(F$,P-1,
   1))<128 THEN 23Ø
24Ø K=K+1:GOTO 22Ø
25Ø PRINT"□ILLEGAL TOKEN□":
   PRINT:GOTO 1ØØ
```

T

For the Tandy change Lines 2Ø and 6Ø and add Lines 45 and 85:

```
2Ø C=43622:FOR K=Ø TO 78
45 IF K=52 THEN C=33155
6Ø C=438Ø2:FOR K=1 TO 34
85 IF K=2Ø THEN C=3331Ø
```

Enter any token and the program will print out the keyword. The numbers for the tokens, in hex, range from 8Ø up to CD for most keywords, and from FF8Ø to FFA1 for functions.

You now only have to discover what the four bytes at the start of the line and the final byte of the line are used for. If you look at the 4th, 18th, 28th and 36th bytes you will notice a pattern . . . 1Ø, 2Ø, 3Ø, 4Ø. These bytes are used to hold the line number and the zero before them is the high order byte used for large line numbers.

The first two bytes of each line are used as a pointer to the byte where the next line starts. For example, the first two bytes are 3Ø and 15 in locations 7681 and 7682.Multiplying the contents of location 7681 by 256 and adding the contents of location 7682, that is 3Ø*256+15, gives 7695 the location which contains the first byte of Line 2Ø. So the final picture looks like this:

```
ptrH  ptrL  Ø     1Ø    PRINT□  "     H
E     L     L     O     "   eol prtH  ptrL
Ø     2Ø    LET   □     A   =   2     eol
ptrH  ptrL  Ø     3Ø    PRINT□  A     eol
ptrH  ptrL  Ø     4Ø    STOP eol Ø    Ø
```

Typing NEW on the Dragon resets not only 7681 and 7682 to Ø but also resets several other locations in memory, and so it is not a simple matter to retrieve the program. You can check this for yourself by using a series of PEEKs to look at the first few location both before and after a NEW. You have to enter the PEEKs one by one—you cannot use a FOR . . . NEXT loop as that sets up a new variable which will alter the program and make it impossible for you to see what's happening.

STORING THE VARIABLES

The last section showed how a program was stored but any variables set up by the program are stored separately in two special variable stores. There are two memory locations—27 and 28—that contain the address of the start of the first store, used for simple variables. Two other locations—29 and 3Ø point to the start of the array variables store. Simple variables can also be pointed to directly using VARPTR followed by the name of the variable. So VARPTR(A) points to the location of the value of the variable A. The *name* of the variable is stored immediately before its value so VARPTR(A)-2 points to the start of its name.

SIMPLE VARIABLES

The next program prints out the first six bytes of the simple variable store showing how variable A is stored:

```
1 A=1
1Ø V=VARPTR(A)-2
2Ø FOR K=V TO V+6
3Ø PRINTPEEK(K);
4Ø NEXT:PRINT
```

You should see the following numbers (any ASCII codes have been translated for you underneath):

```
65 Ø 129 Ø Ø Ø Ø
A    ←  value  →
```

The letter A is stored first then there's a free byte to leave room for two-lettered variables, followed by the number 1 in floating point format.

STRING VARIABLES

Strings are stored in a similar way to simple variables. Change the A in Line 1Ø to AA$ and change Line 1 to:

```
1 AA$="TESTING"
```

This time you'll get:

```
65 193 7 Ø 3Ø 1Ø Ø
A   A
```

The name AA is stored in the first two bytes and the second letter has 128 added to it to indicate it is a string. The 7 gives the number of bytes allocated (count them). The two zeros are used for garbage collection, that is when the computer clears out all the old strings and tidies up all the current ones. The 3Ø and 1Ø are the high and low bytes of the address where the word TESTING is stored.

If you work it out, this is 3Ø*256+1Ø, which equals 769Ø and is in fact the position of the word in the BASIC program. The Dragon and Tandy save memory this way by not duplicating the word. But as soon as the variable is altered, the new contents are put at the top of available RAM and the pointers are altered to point to the new word. All new strings are stored here at the very top of available RAM.

ARRAY VARIABLES

The next few lines look at the array variable store:

```
1Ø T=1:K=1:V=PEEK(29)*256+PEEK(3Ø)
2Ø T=PEEK(V+2)*256+PEEK(V+3)
3Ø FOR K=V TO V+T-1
4Ø PRINT PEEK(K);
5Ø NEXT
```

Add these lines and RUN the program to get the list of numbers below:

```
1 DIM A(Ø,2)
2 A(Ø,Ø)=1Ø
3 A(Ø,2)=2Ø
```

```
65  Ø  Ø  24  2  Ø  3  Ø  1  132  32  Ø  Ø  Ø
A                            ←   value    →
Ø    Ø    Ø    Ø    Ø    133   32   Ø   Ø   Ø
←      value        →    ←       value       →
```

As usual the first two bytes give the name, and the next two the length (again, count them to check). Then follows the number of dimensions, 2, and two pairs of bytes giving the number of elements set up in reverse order—the Ø3 and Ø1 indicate a 1 by 3 array. After this are the values themselves in floating point format. Notice that space is allocated to element (Ø,1) even though a value is not assigned.

Finally, try a string array:

```
1 DIM A$(2)
2 A$(Ø)="TEXT"
3 A$(2)="ARRAY"
```

```
65 128 Ø 22 1 Ø 3 4 Ø 3Ø 24 Ø Ø Ø Ø Ø Ø 5 Ø
3Ø 41 Ø
```

This is very similar to the way simple strings are stored.

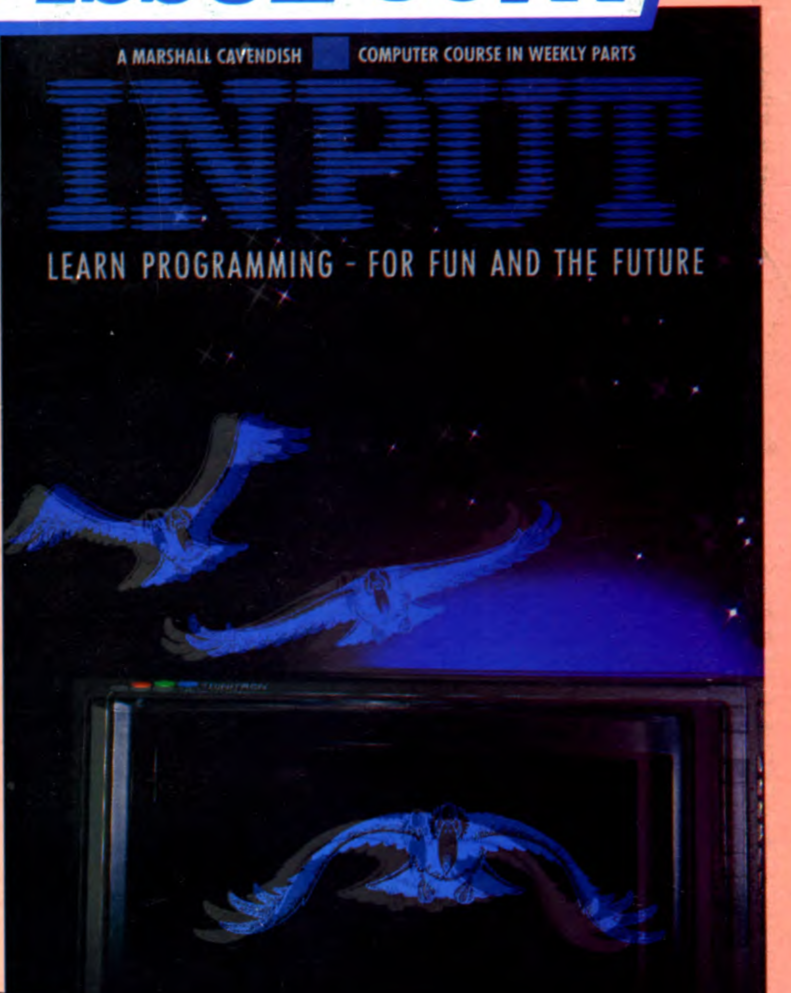
There's the 65 for the A, 128 to indicate a string, two bytes for the length, one for the dimension and one pair giving the number of elements. Next follows three groups of five bytes per string. Each group starts with a number telling how many characters are in that string—4 in the first case for TEXT, then a zero for housekeeping, two bytes giving the address of the string and another byte for housekeeping. This is repeated for the other two strings even the one that hasn't been defined. The 3Ø's may be different if the program is located in a different part of memory.

CUMULATIVE INDEX

An interim index will be published each week. There will be a complete index in the last issue of *INPUT*.

The publishers accept no responsibility for unsolicited material sent for publication in INPUT. All tapes and written material should be accompanied by a stamped, self-addressed envelope.

COMING IN ISSUE 36...


A MARSHALL CAVENDISH
COMPUTER COURSE IN WEEKLY PARTS
INPUT
LEARN PROGRAMMING - FOR FUN AND THE FUTURE

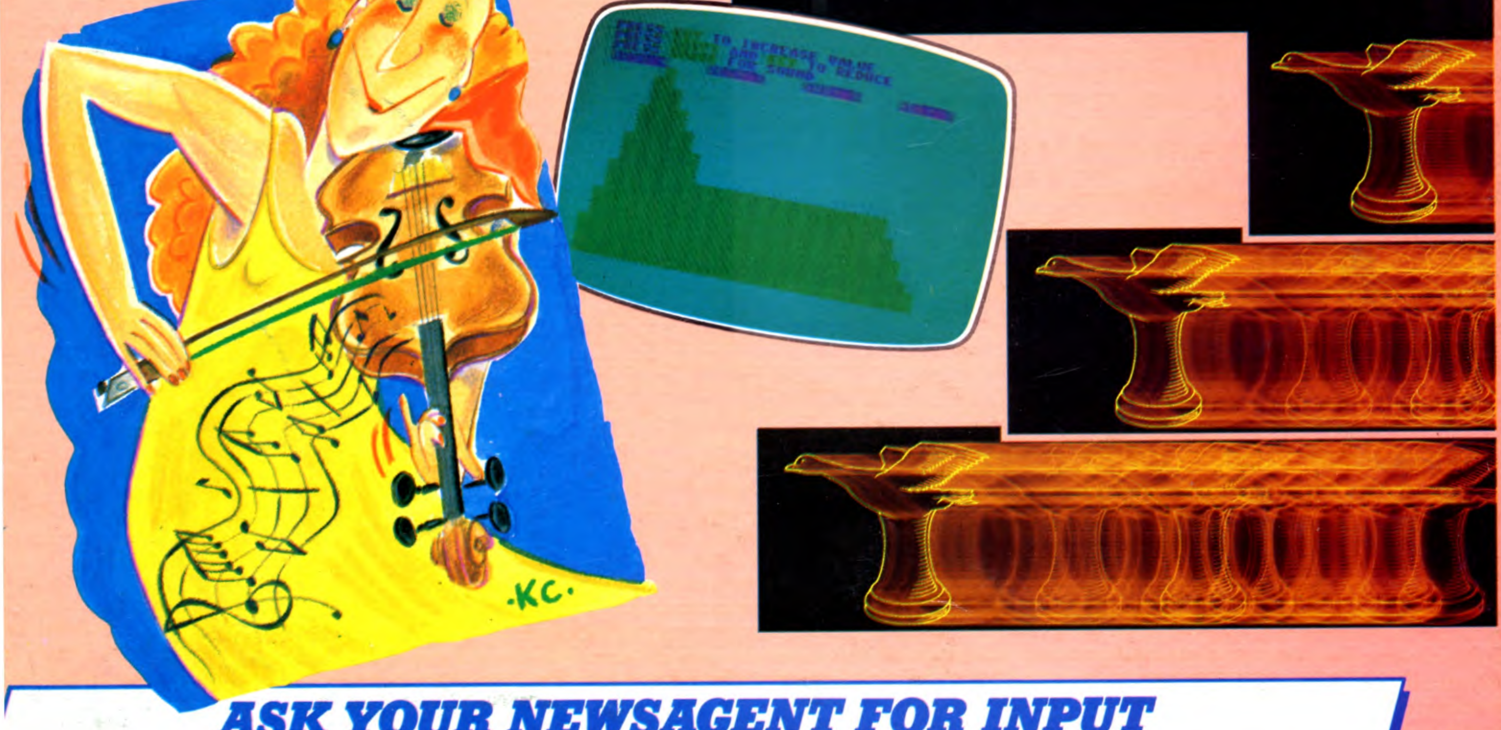
- ☐ *Avert financial disaster (or, at least, see it coming!) with SPREADSHEETS, the cornerstone of forward planning*

- ☐ *Save Willie from the rising tide, waft the clouds along, and gain control over boulders and boas when you RESET THE VARIABLES in CLIFFHANGER*

- ☐ *Put theory into practice and a gaggle of routines to FOX AND GEESE to initialize the game and map future moves*

- ☐ *See practical applications of flicker book animation in MORE ABOUT PAGED GRAPHICS*

- ☐ *Computers always sound the same—or do they? ACORN and COMMODORE owners can imitate real sounds and instruments using SOUND ENVELOPES*



ASK YOUR NEWSAGENT FOR INPUT