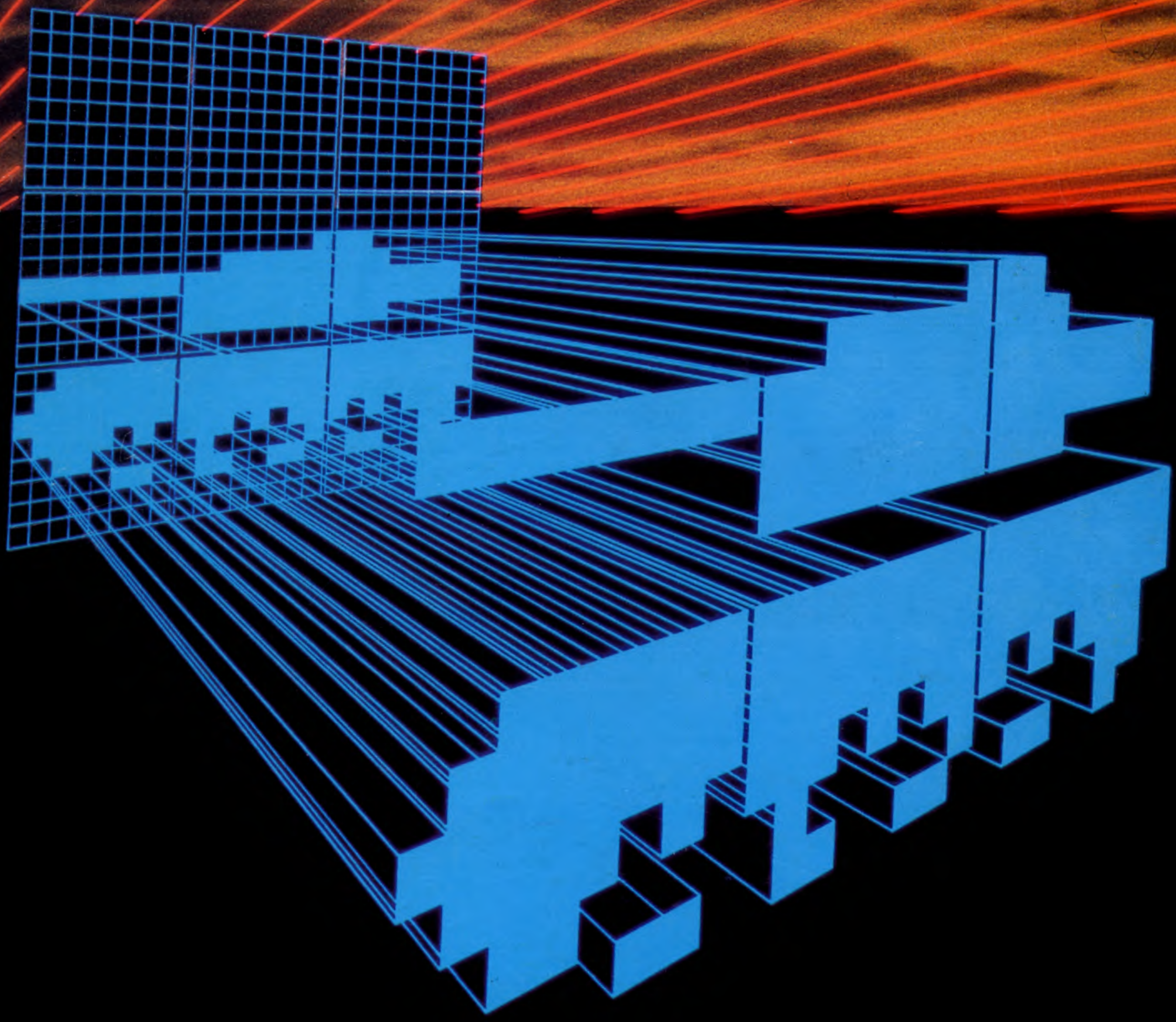
A MARSHALL CAVENDISH 25 COMPUTER COURSE IN WEEKLY PARTS

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. 2 **No 25**

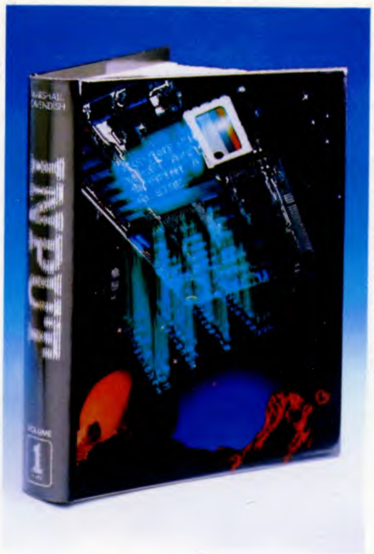
INDEX

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, Paul Chave. Pages 765–769, Paul Chave, Ian Stephen. Pages 770, 771, Chris Lyon. Page 772, Steve Bielschowsky. Page 775, Kevin O'Brien. Pages 776, 777, Ellis Nadler, Funny Business. Pages 784–789, Paul Chave. Pages 790, 791, Ian Stephen. Pages 792, 793, Will Stephen. Pages 794, 795, Grant Symon.

Published by Marshall Cavendish Partworks Ltd, 58 Old Compton Street, London W1V 5PA, England. Printed by Artisan Presss, Leicester and Howard Hunt Litho, London.



I here 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:

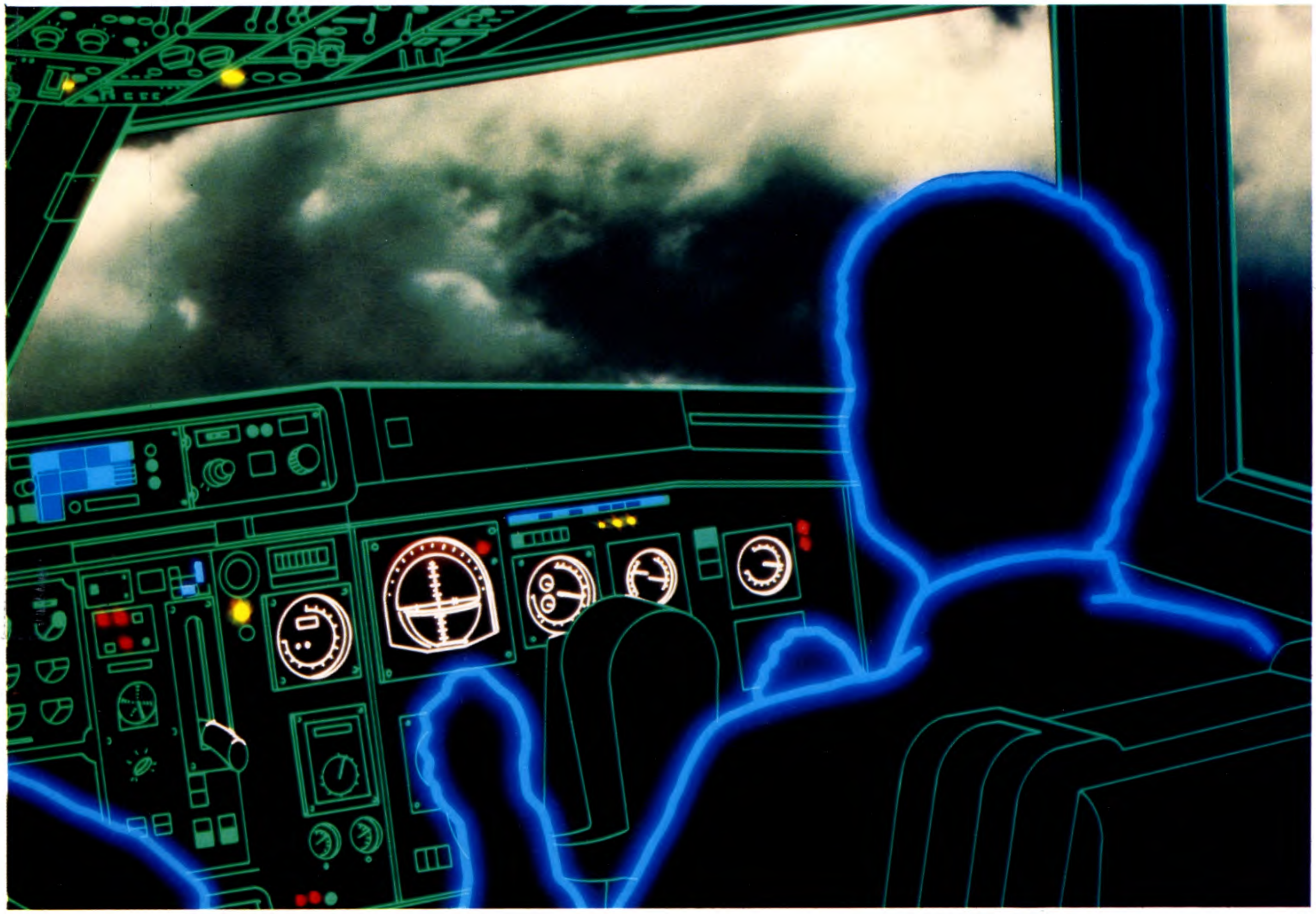
The SINCLAIR ZX SPECTRUM (16K, 48K, 128 and +), COMMODORE 64 and 128, ACORN ELECTRON, BBC B and B+, and the DRAGON 32 and 64.

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

THE FINAL APPROACH

- A KEYBOARD CONTROL ROUTINE
- CHECKING FOR LANDING ON THE RUNWAY
- CRASHING
- DAMAGE ESTIMATES



Turn off the auto-pilot, but don't breathe a sigh of relief yet—you're in control. Using just six keys, you must bring the aeroplane safely down to earth

In this third and final part of the flight simulator program, you'll at last gain control of the aeroplane.

Three pairs of keys allow you to increase and decrease the engine speed (revs), turn the aeroplane, or climb and descend.

You'll find yourself 20,000 metres from the runway and 2000 metres high. Guiding the aeroplane in isn't easy, particularly if you are an inexperienced pilot. Don't expect to become expert in your first few flights—piloting an aircraft is a skill that comes to few people naturally.

As you persevere and become more expert at controlling the aeroplane, your landings will improve. After each landing (or crash!) you will get a report on the Final Flight Details. Study these to see where you went wrong, and try to make up for the deficiencies in your control.

```
3000 LET POW=0: LET K$=INKEY$: IF
   K$="" THEN RETURN
3010 IF K$="S" THEN LET POW=-1
3020 IF K$="F" THEN LET POW=1
3030 IF K$="Q" THEN LET PT=PT+1
3040 IF K$="A" THEN LET PT=PT-1
3050 IF K$="O" AND RL>-30 THEN LET
   RL=RL-1
3060 IF K$="P" AND RL<30 THEN LET
   RL=RL+1
3070 RETURN
5020 CLS: INPUT "INPUT WIND SPEED
   (1-50) MPH",XØ
5025 IF XØ>50 OR XØ<1 THEN GOTO 5020
5030 INPUT "WIND DIRECTION (Ø-359)
   DEGREES",X1
5035 IF X1>359 OR X1<Ø THEN GOTO
   5030
5040 LET XØ=XØ/3
```

```
5Ø5Ø PRINT '"WIND
   SPEED=□";3*XØ;"□M/S": PRINT
   "DIRECTION=□";X1;"□DEGREES"
5Ø55 PAUSE 1ØØ: CLS
5Ø6Ø LET WY=—XØ*COS (X1*C)
5Ø7Ø LET WX=—XØ*SIN (X1*C)
55ØØ GOSUB 3ØØØ: GOSUB 1ØØØ
551Ø IF PZ<=Ø THEN GOTO 6ØØØ
552Ø GOSUB 2ØØØ
553Ø GOTO 55ØØ
6ØØØ IF ABS RL>RT OR PT>TP OR PT<Ø
   OR AS>8Ø THEN GOTO 6Ø3Ø
6Ø1Ø IF ABS PX>WR OR ABS PY>1ØØØ
   THEN GOTO 6Ø6Ø
6Ø2Ø CLS: PRINT "□CONGRATULATIONS
   ON A SUCCESSFUL LANDING": GOTO
   61ØØ
6Ø3Ø FOR N=Ø TO 2Ø STEP .5: PLOT
   127,13Ø: DRAW 12Ø—INT
   (RND*24Ø),45—INT (RND*9Ø): BEEP
   .ØØ5,2Ø—N: NEXT N
6Ø4Ø PAUSE 5Ø
6Ø5Ø CLS : PRINT "A CRASH LIKE THAT HAS
   WRECKED□□□THE AIRCRAFT AND
   KILLED THE□□□□□PASSENGERS!":
   GOTO 61ØØ
6Ø6Ø CLS : PRINT "YOU LANDED OFF THE
   RUNWAY"
6Ø7Ø IF AS<4Ø THEN PRINT "FORTUNATELY
   YOU WEREN'T GOING□□□FAST
   ENOUGH TO DO MUCH DAMAGE": GOTO
   61ØØ
6Ø8Ø IF AS<8Ø THEN PRINT "AT THAT
   SPEED YOU GOT AWAY WITH LIGHT
   DAMAGE AND A FEW BRUISES": GOTO
   61ØØ
6Ø9Ø PRINT "MISSING THE RUNWAY AT
   THAT SPEED HAS LEFT NO SURVIVORS!"
61ØØ PRINT '"FINAL FLIGHT DETAILS"
611Ø PRINT '"AIRSPEED=□";INT
   AS;"□M/S": PRINT
   "DISTANCE=□";INT (SQR
   (PY*PY+PX*PX)): PRINT
   "PITCH□□□=□";PT
612Ø PRINT "ROLL□□□□=□";RL:
   PRINT "RPM□□□□□=□";INT
   (1Ø*TC)/1Ø;"□X□1ØØØ"
613Ø PRINT "DRIFT□□□=□";INT ABS
   PX;"□MTRS": PRINT
   "BEARING□=□";AD;"□DEGREES"
614Ø PRINT '"DO YOU WANT ANOTHER GO
   (Y/N)?"
615Ø LET A$=INKEY$: IF A$<>"Y" AND
   A$<>"N" THEN GOTO 615Ø
616Ø IF A$="N" THEN CLS : STOP
617Ø RUN
```

This time, the first section of the program—the subroutine from Line 3ØØØ to 3Ø7Ø—gives you control over the 'plane.

Using INKEY$ to detect the keypresses, you can change the revs, climb, and bank. A new variable, POW, is used in Lines 3Ø1Ø and 3Ø2Ø to change the engine revs—see Lines 2225 and 2226 on page 734. POW is set to zero in Line 3ØØØ each time the subroutine is called. The S key is used to slow down the revs and the F key is used to go faster.

Q and A are used to change the rate of climb by incrementing or decrementing the variable PT in Lines 3Ø3Ø and 3Ø4Ø. Finally, you can use O and P to bank the aircraft—Lines 3Ø5Ø and 3Ø6Ø look after this, altering the variable RL accordingly.

Lines 5Ø2Ø to 5Ø7Ø allow you to vary the level of difficulty of the simulated landing by giving you the option of having to contend with a wind of various speeds, and to choose a direction for it. The easiest option is a wind speed of 1 m/s at a direction of 0 degrees. From your choices, Line 5Ø6Ø works out the speed of the wind in the forward direction, and Line 5Ø7Ø calculates the wind speed from left to right. SIN and COS are used to work out the components of the speed vector in each direction (see pages 584 to 592). WX and WY are used by the program to alter the position of the aircraft—GX and GY in Line 5Ø8Ø.

The core of the program is the four lines from Line 55ØØ to 553Ø, which call all the important subroutines in turn so that the aircraft's position is updated continuously—you have already entered these routines.

Line 551Ø tests if the aircraft has touched down by checking whether the height parameter, PZ, has reached zero or below. The program jumps out of the 'airborne' loop, and enters the routine which checks if the aircraft has been landed successfully, or not, as the case may be.

In this routine, Line 6ØØØ checks if the roll is within the roll tolerance, or the pitch is less than the pitch tolerance, or the bearing is greater than the yaw tolerance, or the airspeed is greater than 8Ø m/s. If any of these conditions is satisfied, the aircraft has crashed. This is because it has landed with one wing too low, nose down, at an angle across the runway, or simply too fast. The program jumps to Line 6Ø3Ø—the crash routine. Line 6Ø3Ø draws cracks radiating from the top centre of the cockpit window. As each crack is drawn, there's a BEEP, too. After the crash, there's a pause in Line 6Ø4Ø before Line 6Ø5Ø tells the pilot the bad news, and then displays all the flight details.

If the aircraft hasn't crashed, Line 6Ø1Ø checks the position that it landed. If the machine has landed off the runway, the program jumps to Line 6Ø6Ø which tells the pilot what has happened. If the airspeed is below 5Ø m/s, Line 6Ø7Ø tells the pilot that the 'plane has landed without being wrecked. Similarly, Lines 6Ø8Ø and 6Ø9Ø tell you whether the 'plane crash resulted in light damage and a few bruises, or if it was a total wreck with no survivors—now start searching for the black box.

The flight simulator now needs only these few lines to be complete. Don't forget to use either a Simons' BASIC cartridge, or the complete *INPUT* high-res routine, though.

```
3ØØØ GET K$:IF K$="" THEN RETURN
3Ø1Ø IF K$="S" AND TC>.2 THEN
   TC=TC—.2
3Ø2Ø IF K$="F" AND TC<8.8 THEN
   TC=TC+.2
3Ø3Ø IF K$="Q" THEN PT=PT+1
3Ø4Ø IF K$="A" THEN PT=PT—1
3Ø5Ø IF K$="O" AND RL>—3Ø THEN
   RL=RL—1
3Ø6Ø IF K$="P" AND RL<3Ø THEN
   RL=RL+1
3Ø7Ø RETURN
55ØØ GOSUB 2ØØØ:GOSUB 3ØØØ:GOSUB
   1ØØØ
551Ø IF PZ<=Ø THEN 6ØØØ
552Ø GOSUB 2ØØØ:GOSUB 2Ø6Ø
553Ø GOTO 55ØØ
6ØØØ IF ABS(RL)>RT□OR PT>TP□OR
   PT<Ø OR AS>8Ø THEN 61ØØ
6Ø1Ø IF ABS(PX)>WR□OR ABS(PY)>
   1ØØØ THEN 62ØØ
6Ø2Ø PAUSE 1:PRINT
   "♥CONGRATULATIONS A SUCCESSFUL
   LANDING":GOTO 65ØØ
61ØØ FOR Z=1 TO 15:LINE 8Ø,55,
   RND(1)*16Ø,RND(1)*11Ø,RND(1)*3+1:
   COLOUR 6,Z
611Ø NEXT Z: PAUSE 3
612Ø PRINT "♥A CRASH LIKE THAT HAS
   WRECKED THE"
613Ø PRINT "AIRCRAFT AND KILLED YOUR
   PASSENGERS":GOTO 65ØØ
62ØØ PAUSE 1:PRINT "♥YOU LANDED OFF
   THE RUNWAY"
621Ø IF AS<4Ø THEN PRINT "FORTUNATELY
   YOU WEREN'T GOING FAST"
6215 IF AS<4Ø THEN PRINT "ENOUGH TO
   DO MUCH DAMAGE":GOTO 65ØØ
622Ø IF AS<8Ø THEN PRINT "AT THAT
   SPEED YOU GOT AWAY WITH LIGHT"
6225 IF AS<8Ø THEN PRINT "DAMAGE AND
   A FEW BRUISES":GOTO 65ØØ
623Ø PRINT "MISSING THE RUNWAY AT
   THAT SPEED HAS□□□□LEFT NO
   SURVIVORS !"
65ØØ COLOUR 6,6:NRM:PRINT "▣▣▣
   ▣▣▙▙▙▙FINAL FLIGHT
   DETAILS"
```

```
6510 PRINT "▣AIRSPEED□=";
   INT(AS);"M/S"
6515 PRINT "DISTANCE□=";INT
   (SQR(PY*PY+PX*PX)):PRINT
   "▙▙▙PITCH□=";PT
6520 PRINT "▙▙▙▙ROLL□=";
   RL:PRINT "▙▙▙▙▙RPM□=";
   INT(10*TC)/10;"X□1000"
6530 PRINT "▙▙▙DRIFT□=";INT
   (ABS(PX));"METRES":PRINT
   "▙BEARING□=";AD;"DEGREES"
6540 PRINT "▣▣DO YOU WANT ANOTHER
   GO (⇧Y/N◪) ?":FLASH 5,10:POKE
   650,0
6550 GET A$:IF A$<>"Y" AND
   A$<>"N" THEN 6550
6560 OFF:IF A$="N" THEN PRINT
   "♥■":COLOUR 6,1:END
6570 RUN
```

The program's main loop, from Lines 5500 to 5530, has been changed so that the control subroutine can be utilized.

Lines 3000 to 3070 give you keyboard control over the aeroplane. Lines 3010 and 3020 allow you to use S and F to make the aeroplane go slower or faster—S and F alter the revs by incrementing or decrementing TC. Q and A alter the pitch of the aircraft. In other words, pressing Q will make the aeroplane descend, and pressing A will make the aeroplane climb by incrementing or decrementing PT. Similarly, Lines 3050 and 3060 O and P alter the aeroplane's roll—in other words, O and P allow you to turn the aeroplane.

If PZ has become less than or equal to zero in Line 5510, the aeroplane has touched down and the program jumps to Line 6000. The Lines from 6000 to 6540 look at where you've landed it and how well.

Line 6000 checks if the roll and pitch are within the allowable limits, whether the bearing is correct, and if the airspeed is below 80 metres per second. If the speed or orientation of the aircraft is outside the range permitted, the program jumps to Line 6100 which draws a series of cracks in the cockpit window. Following a short PAUSE, Lines 6120 and 6130 imply an investigation and possible prosecution by the Civil Aviation Authority—in other words, you crashed.

Line 6010 checks the position of the aeroplane relative to the runway. Line 6200 displays the bad news: YOU LANDED OFF THE RUNWAY. If the airspeed at that time was less than 40 metres per second, Lines 6210 and 6215 tell the pilot FORTUNATELY YOU WEREN'T GOING FAST ENOUGH TO DO MUCH DAMAGE. The program jumps to Line 6500

If the speed was between 40 and 80 metres per second, the message AT THAT SPEED YOU GOT AWAY WITH LIGHT DAMAGE AND A FEW BRUISES. If the speed was greater than 100 metres per second, it's a case for the Civil Aviation Authority again.

If the aeroplane has landed correctly, the computer reaches Line 6020 which displays the successful landing message. Whatever the outcome of the touchdown, the next stop is the Flight Details routine starting at Line 6500.

Line 6500 displays the flight details title before Lines 6510 to 6530 display the values of all the variables relating to the aeroplane. This is where you can judge exactly how you've done.

At the end of the program there's an ANOTHER GO? routine so you can try again.

```
1080 IF INKEY(-99) AND TC>.2 THEN
   TC=TC-.2
1090 IF INKEY(-74) AND TC<8.8 THEN
   TC=TC+.2
1100 IF INKEY(-56) THEN PT=PT+1
1110 IF INKEY(-87) THEN PT=PT-1
1120 IF INKEY(-98) AND RL>-30 THEN
   RL=RL-1
1130 IF INKEY(-67) AND RL<30 THEN
```

```
   RL=RL+1
1180 CLS:INPUT"ENTER WIND SPEED (1-50)
   M/S AND DIRECTION (0-359)
   DEGREES□",X0,X1
1190 IF X0>50 OR X0<1 OR X1>359 OR
   X1<0 THEN 1180
1320 IF ABS(RL)>RT□OR PT>TP□OR
   PT<0 OR AS>80 THEN 1350
1330 IF ABS(PX)>WR□OR ABS(PY)>
   1000 THEN 1380
1340 CLS:PRINT '"CONGRATULATIONS A
   SUCCESSFUL LANDING":GOTO 1420
1350 GCOL0,3:MOVE 100,500:DRAW
   100+RND(500)+310,500+RND(300)
   +50:DRAW1180,900:MOVE 100,900:
   DRAW 100+RND(500)+310,500+RND
   (300)+50:DRAW1180,500
1360 FOR K=1 TO 4000:NEXT:PRINT
1370 CLS:PRINT"A CRASH LIKE THAT HAS
   WRECKED THE□□□□□□□□
   AIRCRAFT AND KILLED ALL YOUR
   PASSENGERS":GOTO 1420
1380 CLS:PRINT"YOU LANDED OFF THE
   RUNWAY"
1390 IF AS<40 THEN PRINT"FORTUNATELY
   YOU WEREN'T GOING FAST□□□□
   □□ENOUGH TO DO ANY
   DAMAGE":GOTO 1420
1400 IF AS<80 THEN PRINT"AT THAT
   SPEED YOU GOT AWAY WITH
   LIGHT□□□DAMAGE AND A FEW
   BRUISES":GOTO 1420
1410 PRINT"MISSING THE RUNWAY AT THAT
   SPEED HAS□□□□LEFT NO
   SURVIVORS !"
1420 PRINT'"FINAL FLIGHT DETAILS"
1430 PRINT'"AIR SPEED□□□□
   □□=□";INT(AS);"□M/S":
   PRINT"DISTANCE□□□□□□□□
   =□";INT(SQR(PY*PY+PX*PX)):
   PRINT"PITCH□□□□□□□□□□□
   =□";PT
1440 PRINT"ROLL□□□□□□□□□
   □□□=□";RL:PRINT"RPM
   □□□□□□□□□□□□□□=□";
   INT(10*TC)/10;" X 1000"
1450 PRINT"DRIFT□□□□□□□□
   □□□=□";INT(ABS(PX));
   "□METRES":PRINT"BEARING
   □□□□□□□□□□=□";AD;"□
   DEGREES"
1460 PRINT'"DO YOU WANT ANOTHER GO
   (Y/N) ?"
1470 A=GET:IF A=78 THEN END
1480 IF A=89 THEN RUN ELSE 1470
```

Lines 1080 to 1130 give you control over the aeroplane. Pressing the RETURN key will increase the revs, while pressing the space bar will decrease the revs by incrementing or decrementing the TC variable. P and L control the pitch, or the climb and descent—the variable PT. The aeroplane can be turned by changing the roll—press Z or X.

Lines 1180 and 1190 allow the pilot to choose windspeed and direction against which to land so that the level of difficulty can be varied. Line 1180 is the prompt, and Line 1190 is a check that the numbers chosen are within the permitted range.

The remainder of the program is concerned with the possible outcomes of touching down—has the aeroplane been landed successfully, or have all the passengers been killed in a total wreck? Line 1320 checks if the roll, pitch and speed are within the limits which will allow a successful landing. If any of these limits is exceeded, the program jumps to Line 1350, the crash routine, displaying up a cracked window before Line 1370 tells the pilot the bad news. The program jumps to the flight details routine at Line 1420 onwards.

Line 1330 checks if the aeroplane has landed on the runway. If it hasn't, Line 1380 tells the pilot the bad news. Landing off the runway doesn't automatically wreck the aeroplane—if it was going slowly enough there may be minimal damage. Lines 1390 and 1400 check the airspeed at landing, and report the level of damage. If the speed was over 100 metres per second, Line 1410 tells the pilot MISSING THE RUNWAY AT THAT SPEED HAS LEFT NO SURVIVORS—the newsmen are on their way!

The Final Flight Details are displayed on the screen by Lines 1420 to 1450. The values of each of the variables relating to the aeroplane are PRINTed as a guide to the level of the pilot's expertise.

For those of you who haven't tired of the slaughter of poor innocent passengers, Lines 1460 to 1480 are an ANOTHER GO? routine.

Before you RUN the program, delete Line 5505. Tandy owners should also change Line 3000 to REM, change 239 in Line 3010 to 251, change 251 in Line 3020 to 254, and change 223 in Lines 3030 and 3060 to 247.

```
3000 IF PEEK(337)=255 THEN RETURN
3010 IF PEEK(341)=239 AND TC>.2 THEN
   TC=TC-.2
3020 IF PEEK(344)=251 AND TC<8.8 THEN
   TC=TC+.2
3030 IF PEEK(341)=223 THEN PT=PT+1
3040 IF PEEK(342)=223 THEN PT=PT-1
3050 IF PEEK(343)=223 AND
   RL>-30THEN RL=RL-1
3060 IF PEEK(344)=223 AND RL<30THEN
   RL=RL+1
3070 RETURN
5020 CLS:INPUT"INPUT WIND SPEED (1-50)
   M/S";X0
5025 IF X0>50 OR X0<1 THEN 5020
5030 INPUT"WIND DIRECTION (0-359)
   DEGREES";X1
5035 IF X1>359 OR X1<0 THEN5030
5040 X0=X0/3
5050 PRINT:PRINT"WIND SPEED=□";
   3*X0;"M/S":PRINT"DIRECTION=□";
   X1;"DEGREES"
5060 WY=X0*COS(X1*C)
5070 WX=-X0*SIN(X1*C)
5500 GOSUB3000:GOSUB1000
5510 IFPZ<=0 THEN6000
5520 GOSUB2000
5530 GOTO5500
6000 IF ABS(RL)>RT OR PT>TP OR PT<0
   OR AS>80 THEN 6100
6010 IF ABS(PX)>WR OR ABS(PY)>1000
   THEN6200
6020 CLS:PRINT"□CONGRATULATIONS
   A SUCCESSFUL□□□□LANDING":
   GOTO6500
6100 LINE(0,0)-(RND(128)+63,
   RND(96)+48),PSET:LINE-(255,
   191),PSET:LINE(255,0)-(RND
```

```
   (128) + 63,RND(96) + 48),PSET:
   LINE – (Ø,191),PSET
611Ø FORK = 1TO2ØØØ:NEXT
612Ø CLS:PRINT"□A CRASH LIKE THAT
   HAS WRECKED□□□THE AIRCRAFT
   AND KILLED YOUR□□□□
   PASSENGERS":GOTO65ØØ
62ØØ CLS:PRINT"□YOU LANDED OFF THE
   RUNWAY"
621Ø IF AS < 4Ø THEN PRINT
"□FORTUNATELY YOU WEREN'T
   GOING□□□FAST ENOUGH TO DO
   MUCH□DAMAGE":GOTO65ØØ
622Ø IF AS < 8Ø THENPRINT"□AT THAT
   SPEED YOU GOT AWAY WITH LIGHT
   DAMAGE AND A FEW
   BRUISES":GOTO65ØØ
623Ø PRINT"□MISSING THE RUNWAY AT
   THAT□□□□□□SPEED HAS LEFT NO
   SURVIVORS !"
65ØØ PRINT:PRINT"□FINAL FLIGHT
   DETAILS"
651Ø PRINT:PRINT"□AIRSPEED□ = ";
   INT(AS);"M/S":PRINT"□DISTANCE
   □ = ";INT(SQR(PY*PY + PX*PX)):
   PRINT"□PITCH□□□□ = ";PT
652Ø PRINT"□ROLL□□□□□ = ";RL:
   PRINT"□RPM□□□□□□ = ";INT
   (1Ø*TC)/1Ø;"□X□1ØØØ"
653Ø PRINT"□DRIFT□□□□ = ";INT
   (ABS(PX));"METRES":PRINT
   "□BEARING□□ = ";AD;"DEGREES"
654Ø PRINT:PRINT"□DO YOU WANT
   ANOTHER GO (Y/N) ?"
655Ø A$ = INKEY$:IF A$ < > "Y" AND
   A$ < > "N" THEN 655Ø
656Ø IF A$ = "N" THENCLS:END
657Ø RUN
```

The main loop of the program from Line 55ØØ to Line 553Ø now calls the control subroutine which allows you to take control from the manic autopilot. The control subroutine extends from Line 3ØØØ to 3Ø7Ø. Line 3ØØØ checks if no key is being pressed; Lines 3Ø1Ø and 3Ø2Ø read the F and S—faster and slower—keys; Lines 3Ø3Ø and 3Ø4Ø read the up and down arrow keys, to control ascent and descent; and Lines 3Ø5Ø and 3Ø6Ø read the left and right keys which bank the aeroplane to turn it.

Lines 5Ø2Ø to 5Ø7Ø allow you to vary the level of difficulty of the landing by varying the wind direction and speed. The easiest option is a speed 1 m/s at a direction of 0 degrees. From your choices, Line 5Ø6Ø works out the windspeed in the forwards direction, and Line 5Ø7Ø works out the windspeed in the left/right direction. WX and WY are used to alter the position of the aircraft.

If Line 551Ø finds that the aeroplane is on the ground, the program jumps to Line 6ØØØ. Line 6ØØØ checks that the roll, pitch and speed are within acceptable limits. If any one of these is exceeded, Line 61ØØ draws some cracks in the cockpit window and Lines 611Ø to 623Ø display the message.

If the aeroplane's orientation and speed were correct for landing, Line 6Ø1Ø checks whether the aeroplane has landed on the runway. If it has, Line 6Ø2Ø tells the pilot of a successful landing. If it has touched down off the runway, all might not be lost.

If the speed is below 4Ø m/s Line 621Ø tells the pilot that there isn't much damage. If the speed was between 40 and 80 m/s, Line 622Ø tells of light damage and a few bruises. If the speed was any more than 8Ø m/s, it's curtains.

Lines 65ØØ to 653Ø display the values of all the variables connected with the aircraft at the time of landing. Take notice of these, and you should improve.

Finally, Lines 655Ø to 657Ø give you a chance of another go.

GRAPHICS PADS AND TABLETS

'Sketching' with dozens of program instructions isn't a very natural way to make pictures. But with a graphics pad, you can draw as easily as with pencil and paper

One of the areas where computers have fulfilled their early promise—and sometimes exceeded all expectations—has been that of CAD, or computer aided design. Computers of all sizes, from mainframes to small home micros, have been used in all aspects of design. Fashion designers and local builders are now using micros for their work, and the role of computers in design is no longer restricted to grand plans for rebuilding city centres or billion-dollar projects such as the Space Shuttle.

Most home micros have been designed with good graphics capabilities, originally with games in mind, and these capabilities can be exploited by the home user to produce some spectacular results.

One of the most useful peripherals for exploring computer graphics is the *graphics pad*, sometimes called a graphics *tablet* or graphics board. You can sketch on this with a stylus in what is virtually a form of freehand drawing. The movement of the stylus on the surface of the pad is duplicated on the screen by a clever combination of software and hardware.

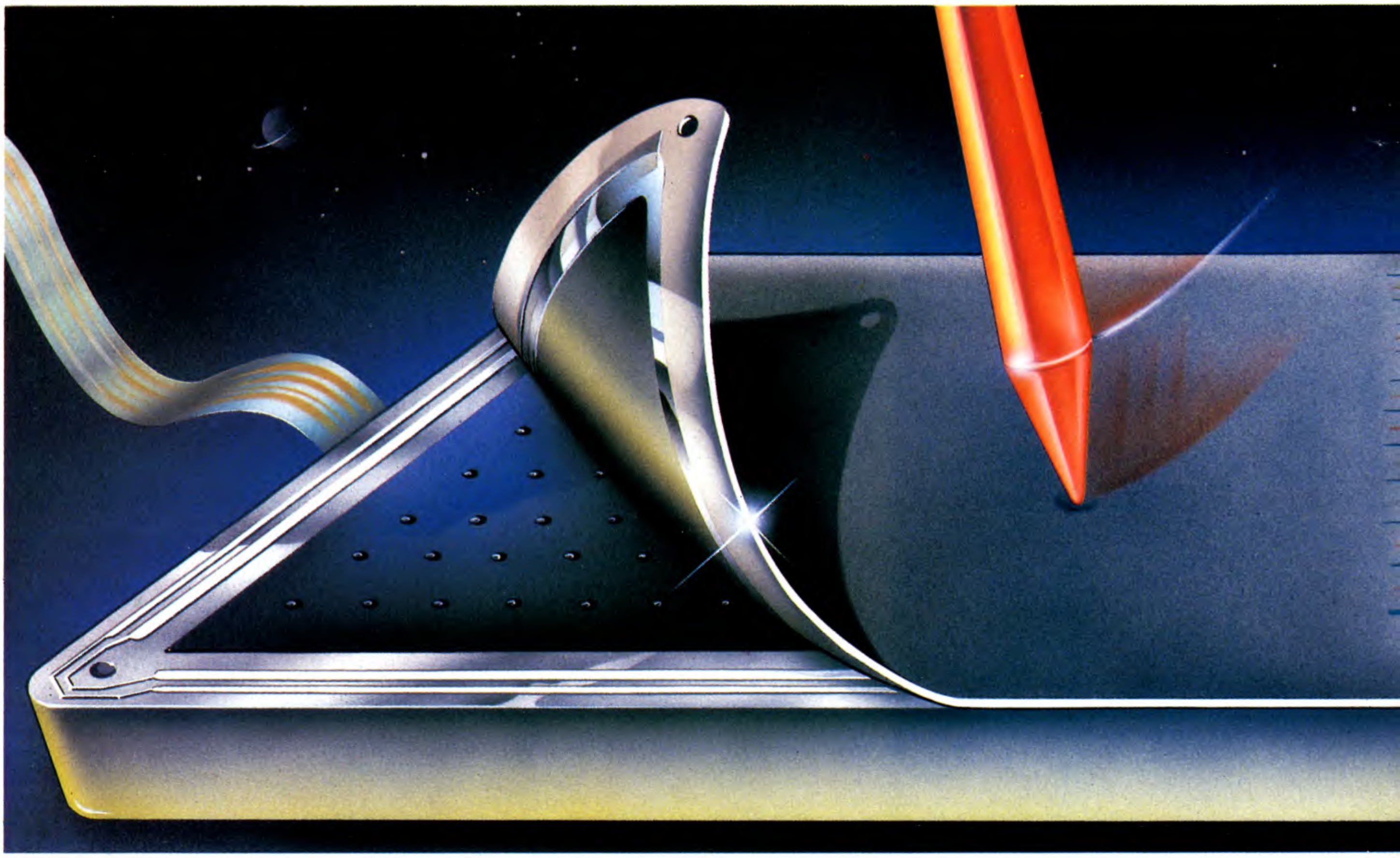
To achieve the equivalent results without a peripheral of this type would involve considerable amounts of programming—and you could not get the same flexibility, nor the ability to amend or erase your creations at will.

Even the simplest graphics pad enables you to draw lines on the screen and then add your own colour to the drawing. Areas can be filled in with colour although, of course, this is limited to colours that the computer is capable of producing. Colours can usually be changed at the touch of a couple of keys or by selection using the pad itself. 'Brush' strokes can also be variable in size so that you can draw with a thick brush or a thin brush, in single or multiple lines—sometimes even with a stippling effect.

PAD FEATURES

Most graphics pads and associated software offer similar facilities as far as drawing is concerned. It's in the manipulation of the drawing that the major differences in their capabilities occur. For example, some graphics pads have a 'zoom' facility which allows you to zoom in to a small area of the drawing to work on it in more detail or to make small corrections—often down to individual pixels.

A good graphics pad will enable you to move parts of a drawing around the screen and to repeat one element of a picture several times. Some allow you to swap around ele-



- THE ADVANTAGES OF PADS
- DIFFERENT TYPES
- DIGITIZATION AND GRID MAPPING
- READING SIGNALS
- DRIVER SOFTWARE
- USING A PAD
- SPECIAL FEATURES
- SAVING YOUR WORKS OF ART

ments of a picture within the same picture and even between different pictures. You may also be able to create mirror images automatically. It's usually possible to SAVE a design, but different graphics pads in conjunction with different micros SAVE the screen in different ways. As a result, there are therefore variations in the facilities available with each pad. Some enable you to use the SAVEd design in other programs—so you can create a background for a game you have written, for example. Other pads are not so versatile.

HARDCOPY

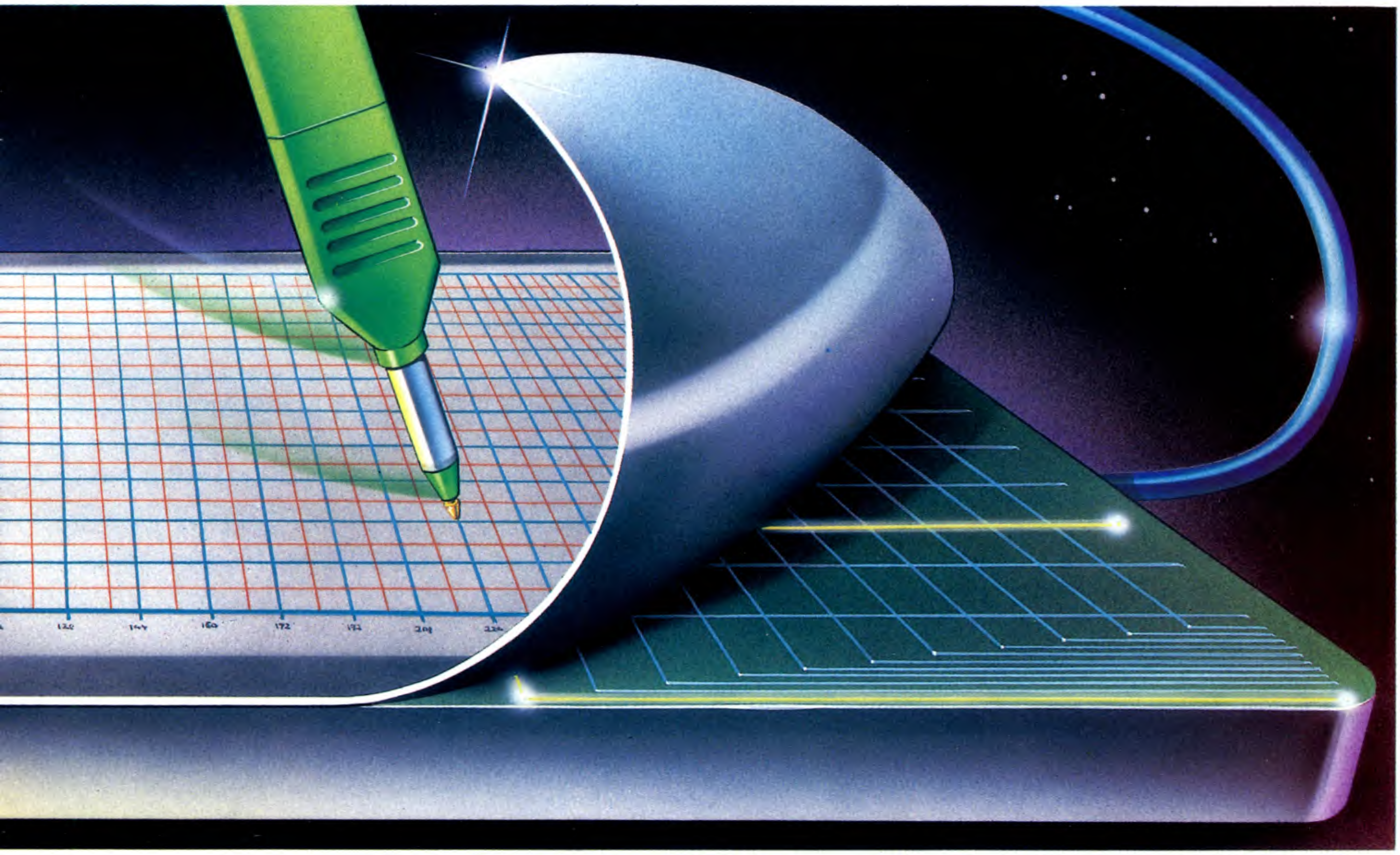
Pretty screen pictures which are used only in isolation for displays are not really, in the long run, of very much use. Most home computers can be programmed to *dump* a screen picture to a suitable printer and this facility should be available on the graphics pad if you do want some form of hardcopy. Even though this is unlikely to be in more than one colour, different hues can be suggested by use of various methods and styles of hatching and shading.

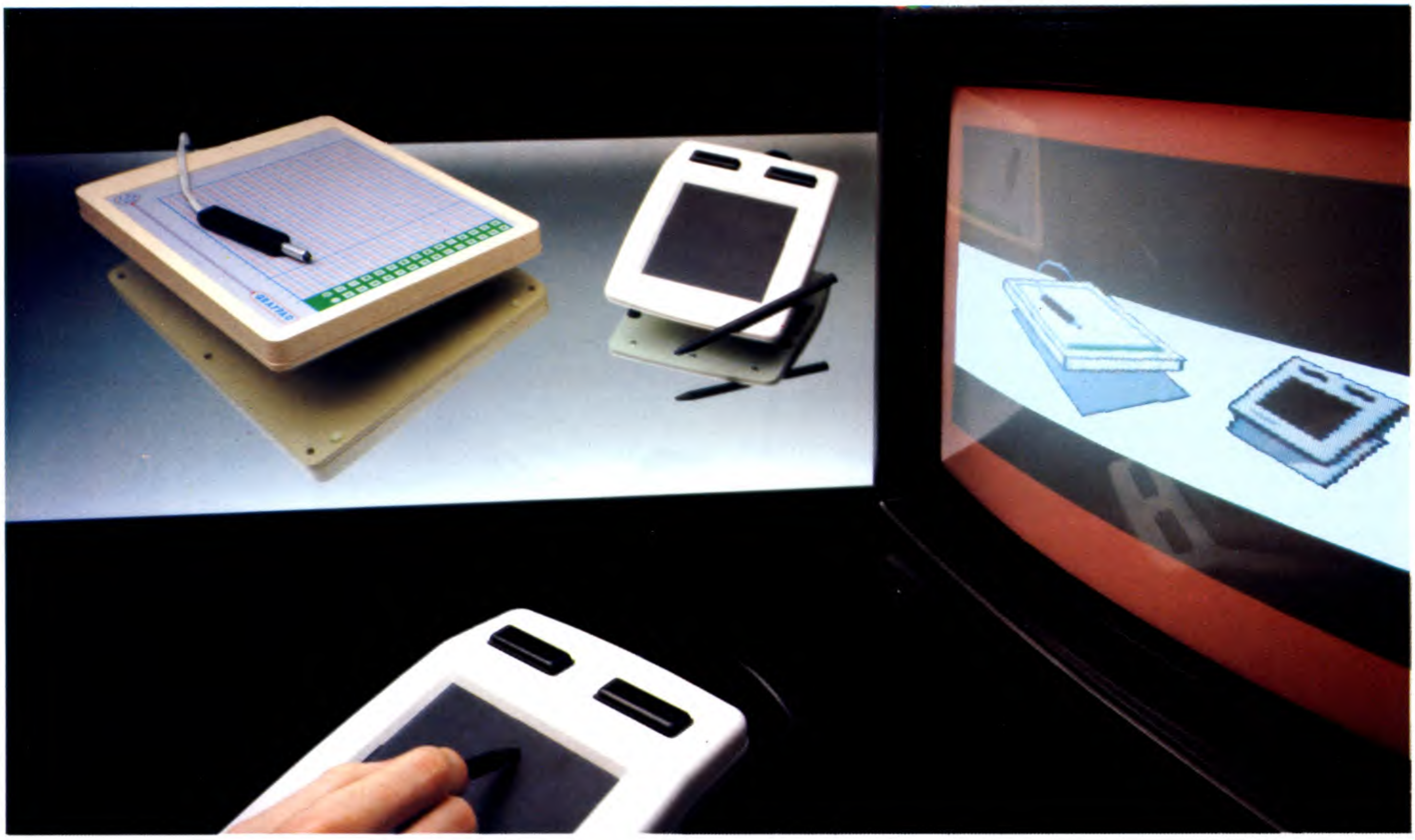
But for the very best results—certainly if you want reasonably good colour copies—you need a plotter—a very sophisticated sort of printer which draws rather than prints. Plotters are often very expensive, and one thing to watch out for is that the manufacturers of your micro or graphics pad may not have envisaged its use with a plotter, so obtaining a full colour hard copy of your screen design may be difficult.

TYPES AVAILABLE

There are several graphics pads for each model of home computer. Popular examples include the Grafpad for the Spectrum, Commodores and BBC; the KoalaPad for the Commodore 64; and the Touchmaster for the Dragon—although new designs are constantly being introduced. The most obvious difference between models is the size of the pad—commonly anywhere from around 100mm × 100mm to 250mm × 300mm—which clearly has a bearing on how easy it is to draw accurately on the surface. Apart from this, these pads differ in the way they work and in what they can actually do—although much of this is due to the software which controls them.

An alternative to the true graphics pad is to fit your computer with one of a number of pantograph-inspired *digital tracers* that are available for each machine. These are usually cheaper than pads, and are supplied from quite a few 'cottage industry' sources—usually by mail-order. These are better for





tracing out intricate shapes from an existing pattern—maps, for example. (However, you can do this sort of work on a pad by improvizing a suitable *overlay*, for example, by tracing the pattern, then laying it on the pad.) But the tracers' construction does not make them very suitable for the sort of freehand drawing that you can do with an unrestricted stylus on a graphic pad.

DIGITIZATION

A graphics pad is actually an example of a digitizer. The principle of digitizing is used extensively in computers since many computing applications are, fundamentally, all about turning an analogue signal into a digital signal. An analogue system signal varies continuously and may take any value within its ultimate limits—an example is an ordinary volume control. But the computer can only understand a digital signal—the difference between off and on, between Ø and 1, between open and closed. Digitization is the process of turning a continuous analogue signal, in which some of the elements may be 'grey' or half on and half off, into a definitive signal which can be understood by computers.

The graphics pad or digitizer turns pictures into numbers. The computer then turns those numbers back into a picture. Without a method of turning pictures into numbers it would be impossible for computers to 'understand' pictures and a great deal of the graphics-related work carried out by computers would be impossible.

The process of digitizing involves dividing a picture up into as many equal parts as possible. The greater the number of parts—usually squares, which give computer-enhanced pictures their characteristic 'bitty' look—the greater the ability to record detail. The number and size of the squares is dependent on the digitizer and the computer. In commercial computer graphics systems, the resolution of the picture is high enough so that it only appears slightly grainy. Many of the pictures in *INPUT* are in fact generated using a digitizing tablet and mainframe computer. The graphics pad for your micro needs to do exactly the same thing, except that the picture is necessarily rather less detailed.

GRID MAPPING

A typical text, or low-resolution, screen on home computers is around 40 columns wide by 25 rows deep—although individual machines vary slightly from this size. This means that the screen is divided into 1000 squares or so. In this situation it would be pointless to use a digitizer which was able to divide a picture into any more than 1000 squares. Although more could be defined, the computer would be unable to use that information. But working with the computer's high resolution screen, which allows pixel-by-pixel definition means that the typical graphics pad can use more squares and hence produce better graphics.

If lines were drawn on the picture to illustrate how it is divided up into squares it would look as if graph paper had been laid over the top—something like the grid on a map. Some pads, for example, the Grafpad, are in fact divided up in exactly this way. In fact, the digitizer, and the graphics pad, use numbers to describe where each square belongs in the same way that we use numbers as map references.

Each square along the top or bottom (the horizontal or X-axis) is numbered and the squares are also numbered down the side (the vertical or Y-axis). On a system where the origin is the lower lefthand corner, square 10/23, for example, is 10 squares in from the side and 23 squares up from the bottom. This means that any position on the picture can be converted into a number which the computer can understand.

Graphics pads use a variety of approaches to the problem of producing a signal that the computer will be able to understand, but they all use some form of *analogue to digital (A–D) converter*.

As the name suggests, this is a form of interface which can take an analogue signal (in this case a constantly variable voltage), and turn it into a digital signal that the computer can understand. The A–D converter does this by gradually increasing its output current step by step until it is one step above the incoming current produced by the graphics pad. The time taken for this to happen gives a number which is then sent to the computer for it to interpret.

Two A–D converters are needed, one for the X-axis and another for the Y-axis. Normally, each number is sent along to the computer in a single byte. That magical number 256 crops up here again, as it so often does in discussions about computers. The eight bits in a byte can represent any number from 0 to 255—256 numbers in all. This means that the grid could conveniently measure 256 squares by 256 squares. Square 0/0 (all the bits in the two bytes set to 0) would represent the top left corner of the graphics pad, and square 255/255 (all the bits in the two bytes set to 1) would represent the bottom right corner. Values in between would specify every other possible position.

The information contained in this signal is then transferred directly onto the screen so the computer doesn't have to store information about the coordinates of each line. If it did need to store these coordinates then the amount of memory needed would be enormous—far more than a home computer could cope with. And you still need room in the memory for the colour plus, of course, the graphics pad program itself.

Most graphics pads are designed so the resolution matches the computer's display, which means that the number of squares on the pad is the same as the pixels on the screen.

PRODUCING SIGNALS

The biggest differences between graphics pads lie in the methods chosen to produce a signal in the first place. Some require a special stylus and rely on contact between the stylus and the active area of the pad to produce a signal, while others are touch sensitive to a stylus, or even finger pressure.

The way that graphic pads work is best understood by looking at the simpler methods of construction, although these are now gradually being superseded.

Perhaps the simplest method of all is to use two sliders, one horizontal and one vertical, to which potentiometers are attached, typically by a pulley arrangement. As each slider moves up and down or across the board, the voltage from each potentiometer changes, giving you unique values for each X and Y position. Another method which also uses potentiometers is the digital tracer, which has an articulated 'arm'. In this system, the stylus head is pivoted from a fixed point. The arm has two joints, one at the 'shoulder' and one at the 'elbow'. Once again, potentiometers at each joint will give a unique combination of voltages for each position on the board. It is this method that is used by the RD Digital Tracer for the Spectrum. Strictly speaking, these 'coordinates' are not X and Y coordinates in the normal sense, because the potentiometers are not operating solely in the X and Y directions.

Touch sensitive graphics pads usually rely on two sheets separated by a small gap, with lines of conducting material painted on the facing surfaces. One sheet carries horizontal lines and the other vertical lines. When pressure is applied two of these 'wires' make contact.

The remaining group of graphic pads are those that produce a signal as a result of interaction between a stylus and wires in the surface of the pad. Included in this group is the Grafpad on which the stylus location is detected by electromagnetism.

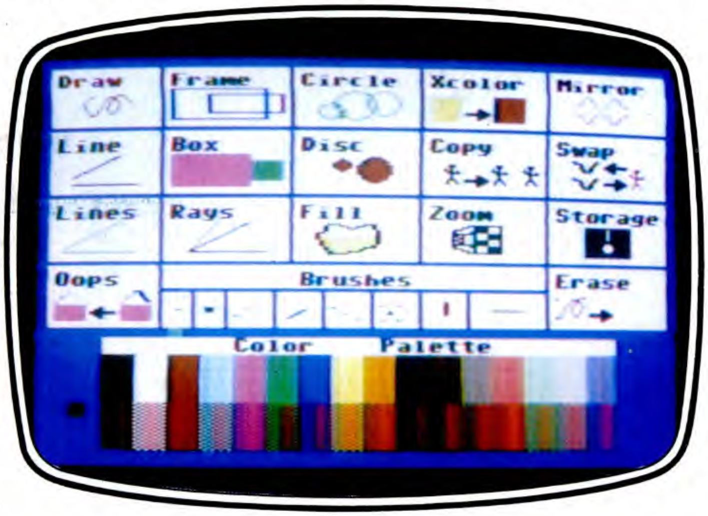
DRIVER SOFTWARE

Despite the obvious hardware differences which any user will appreciate, the software is what actually allows the graphics pad to control the computer. And since it defines not only the available features, but also the ease of use of the pad, the software should, more than anything else, influence your choice. To assess this, you really need to see the system in operation.

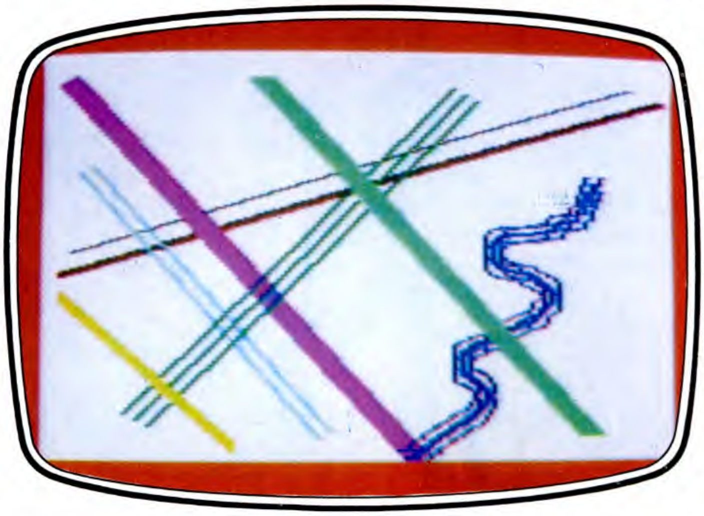
To an extent, the quality of the system must depend upon your computer as well. Obviously, the faster it is and the bigger its memory, the more scope there is for a sophisticated graphics pad. The best of these do require a great deal of memory, which is sometimes in short supply for hi-res applications. Some systems are either cartridge or disk-based, so that swapping data between the computer and its storage system is relatively quick and easy—extending their capabilities by using the storage system as virtual memory.

And of course, the graphics pad software can only make use of the facilities which are available in your machine's hardware. So you can't expect to use a higher definition, or more colours than normal. You might even get less. What a good graphics pad does do is to put all the computer's facilities at your disposal in an easy-to-use way.

There are in fact several computer aided design programs which do this in a very similar fashion, but giving you keyboard



The opening menu of KoalaPad shows available options



Brush 'strokes' and colours can be chosen at will

control rather than pad control. There is an example of such a program in *INPUT*, on pages 566 to 572 and 573 to 577. This gives you a menu-driven choice of drawing commands and other features—each of which can be directed from the keyboard. Graphics pad driver software is very similar, except that the purpose-built pad can be programmed to be far more user-friendly than a jack-of-all-trades keyboard. To see why, it's easiest to look at a graphics pad in operation.

USING A GRAPHICS PAD

Individual pad/computer combinations vary in use for the obvious reasons mentioned above. But they are all designed to work in a way which mimics the natural action of drawing or painting on a board. It is this, more than anything else, which allows them to score over other methods of creating a hi-res image.

Before you can start to create your picture, you first need to specify criteria such as which drawing mode you want, or what colour, for example. This, as with every other option available, is typically displayed as a menu by the program. Menu choices may be selected by pressing buttons, or, with many systems, by using the pad itself as a sort of 'artist's palette'. In this case, the choices may be laid out on screen so that you have, say, a row of ink pots and a row of brush styles, plus various other options. Selection of any of these can be made by moving the stylus to take a screen cursor to the right option, then pressing a button to fix the choice.

Drawing on the pad can be done freehand, using either your imagination or by tracing over an existing picture which you can fix to the pad. In this, as in other applications, the primary job of the software is to interpret the signals from the pad, and to tell the computer that the incoming data refers to a point on the screen. The shape you trace is then instantly displayed.

Freehand drawing can be surprisingly difficult if you want, say, a straight line, as it is easy to wander slightly. You can use drawing aids like rulers, but with many types of pad you need to be careful that these do not trigger the pad instead of the stylus. By far the easiest way to create regular constructions is to use the pad's range of on-board geometric shapes—which may include straight lines, rectangles, circles, arcs, and possibly others, too. These can be selected from the menu, then the stylus is used to fix the position and size of the shape by specifying start and finish points.

At any point in the procedure, there is usually the facility to erase a mistake, although this is limited to the last operations performed since returning to the menu, as the computer can only hold the previous screen in memory. Mistakes which are realized too late for this sort of correction can sometimes be erased by over-drawing in the background colour, although this can be fiddly. In the last resort, you can erase totally and start again.

Ink colour can be changed as necessary when drawing, there is also the facility to fill shapes with colour. This process is usually referred to as 'draw and fill'. The outline is drawn first, then the enclosed space is filled by positioning the cursor within it and making the appropriate selection. Spaces not completely enclosed by lines will tend to leak colour everywhere else. In general, it is best to complete all the outline drawing first, and only then to do your colouring.

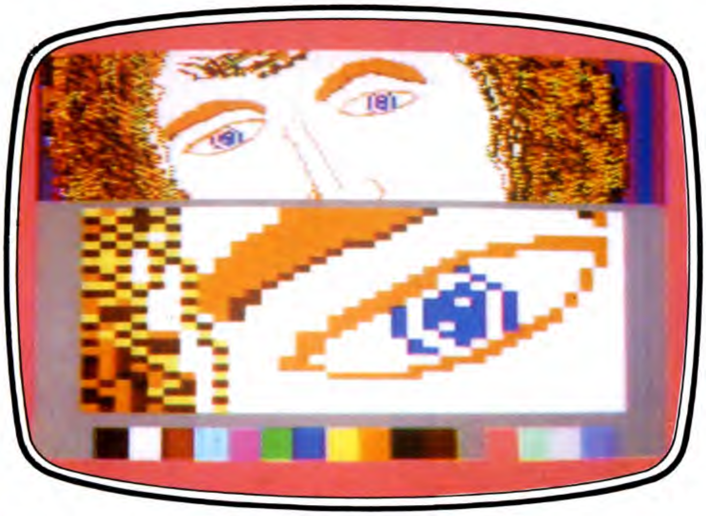
More specialized options vary far more from pad to pad. Examples of these options are the facility to mirror a shape or to copy an image from one part of the screen onto another part—so, for example, you could create a whole forest by drawing just one tree, mirroring it to get an alternative shape, then copying it repeatedly.

There may also be an enlarging facility with which you can take a section of the screen and magnify it. If this is combined with a plotting option, it gives you the chance to edit the image literally pixel by pixel.

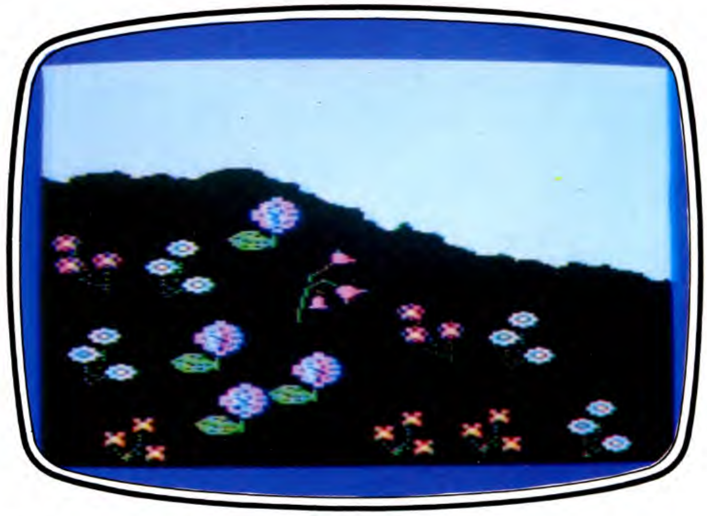
SAVEING THE IMAGE

When you have completed your picture, you will want to keep it. The software also controls the SAVEing of the designs, within the limitations of the way in which your computer SAVEs a screen. (The BBC and Spectrum, for example, SAVE the screen as a whole, while the Commodore 64 in 'multicolour' mode SAVEs in three parts—screen positions, colour and text.) You may be able to save more than one screen within the program for instant recall, but all systems allow you to SAVE pictures to tape or disc. These can usually be recalled into the program for later editing, or even so that you can combine elements from different pictures. Using this facility, some of the software has an on-board set of stock images that you can use for your own picture.

You can keep your masterpiece indefinitely in storage, or load it into another program—a title page for your new game, perhaps? If you want a permanent record, you need access to a colour plotter. Alternatively, you can photograph the screen, being careful to exclude extraneous light which may cause unwanted reflections.



'Zoom' enables you to plot individual pixels

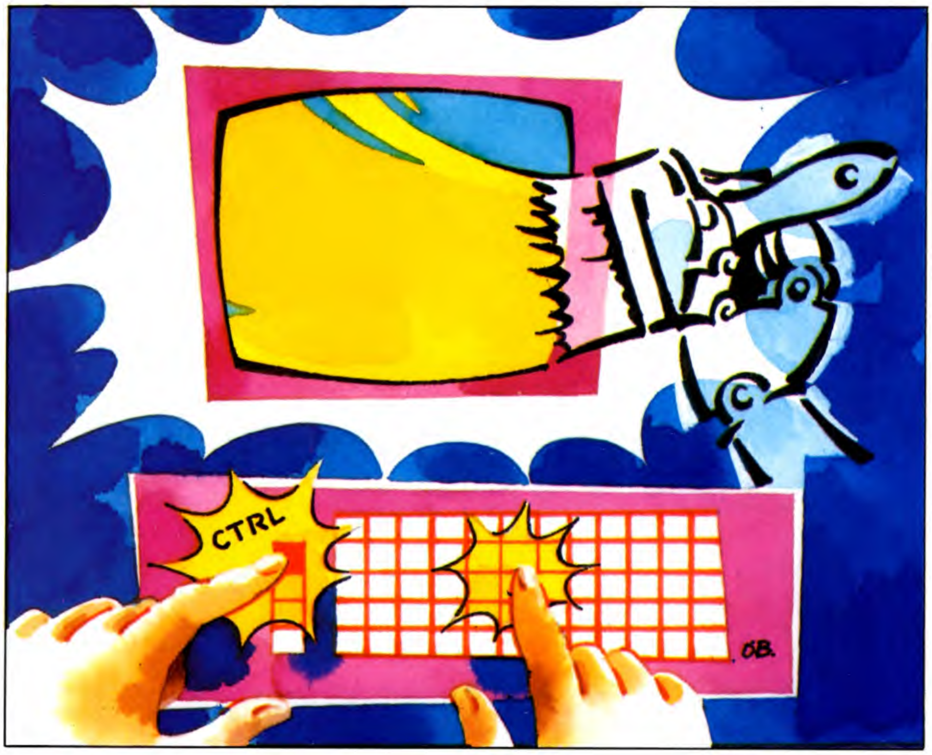


Key elements of a picture can be repeated at will

USING CONTROL CODES

Control codes can be used in place of many BASIC keywords. You can enter them directly from the keyboard and they have the advantage of acting immediately

The control codes are those little-used ASCII codes below 32 and, on the Commodore, a few above 19Ø. These codes can be used for such things as changing colour, moving the cursor and so on. The Dragon and Tandy don't use control codes as the ASCII codes are used to produce the graphics symbols instead.



There is one very good reason to use control codes on the Spectrum, and that is to save memory. The codes can be used in place of PAPER, INK, BRIGHT and FLASH and can be entered by pressing CAPS SHIFT and SYMBOL SHIFT and then a number. These numbers determine which command is actually used:

For PAPER, simply enter the correct colour number from 1 to 7—2 for red for example.

For INK press CAPS SHIFT with the colour number.

For BRIGHT on press 9, and for off press 8.

For FLASH on press CAPS SHIFT and 9 and for off press CAPS SHIFT and 8.

Remember to press CAPS SHIFT and SYMBOL SHIFT first each time.

You can see how much memory is saved in a line such as:

```
1Ø PRINT PAPER 2;INK 6; "MENU"
```

The keywords PAPER and INK and the two semi-colons each take one byte, and each number takes six bytes. The control codes take up only one byte for the pair of shift keys and one for the number. This means a saving of six bytes per command, and in a long program using a lot of colour commands the saving could be enormous.

The commands have to be entered inside the quotes for them to work properly, so in the last example first type:

```
1Ø PRINT "
```

then enter the control codes for PAPER 2 and INK 6 as above, followed by the rest of the line.

The codes take up no space in the listing but have an immediate effect, producing coloured text in the listing as well as on the screen.

In fact you can also use them simply to highlight parts of a listing. In this case you would put them outside any quote marks unless you wanted the colours to appear on the screen as well.

The Commodore computers use control codes all the time as there are no alternative keywords available in BASIC for such things as changing colour, moving the cursor or clearing the screen. The codes appear in listings as graphics symbols or reversed out characters—a reversed out £ (£) sign for red, for example.

These characters are very confusing if you are new to the Commodores, but all the commands that are obtained with the CTRL key are actually printed on the keys. For example, the 3 key has the word RED printed on the front so you know to press CTRL and 3 for red text.

There is a whole list of control codes and their symbols, as these are used in the *INPUT* listings, on page 421.

Control codes on the Acorn can be entered in a variety of ways, either as VDU statements, as PRINT CHR$, or directly from the keyboard using the CTRL key. The article on pages 319 to 320 showed how to use VDU and PRINT CHR$, even in place of some keywords such as COLOUR or MODE. You can produce exactly the same effects with the CTRL key. The numbers used are the same in all cases, although when used with the CTRL key these numbers have to be translated into letters—1 becomes A, 2 becomes B and so on. The manual gives all the conversions. Direct entry of the codes is useful as they act immediately, whether you are in the middle of typing in a program line or running a commercial program such as a wordprocessor.

Some examples are CTRL and B (or VDU2) to turn on a printer, CTRL and C to turn it off, CTRL and U to delete a whole program line, CTRL and N to turn on paged mode, CTRL and O to turn it off.

COMMODORE COLOUR SPRITES

Use this multi-feature sprite generator program to create special sprite data files for use in your own programs. Plus a look at the all-important VIC registers

A sprite is an extremely versatile form of user-defined graphic which gives the Commodore 64 tremendous potential in all forms of games programming.

Though simple in theory, making full use of this powerful facility requires quite a deep knowledge of the workings of the VIC-II chip, particularly of the thirty or so memory locations which affect the physical shape, colour and movement of the standard number of eight sprites.

Many of the general points were discussed earlier (see pages 168 to 172) together with a detailed look at setting up sprites in a single colour. This article shows how to create multicolour sprites, and how to put sprite definitions into action.

You'll also see how to create a number of sprites for boats, sea creatures, a desert island, and other 'characters' that will be used in a later article as the basis of a complete animated scene—the kind of thing on which many commercial games are based.

MULTICOLOUR SPRITES

Multicolour sprites can be defined in a similar way to the single colour sprites covered earlier. And as before, the utility program which follows shortly enables you to work them out easily on the screen.

Multicolour sprites can be made to use up to three colours. You don't get something for nothing—in the process you lose some horizontal resolution, as each pixel is double normal width and two bits of the sprite pattern are needed to define it. Twelve bit-pairs are used to define each of the 21 horizontal lines of a multicolour sprite.

Each of these pairs can take one of four forms: ØØ, Ø1, 1Ø, or 11. Each form is used to give specific information about the particular double-width pixel it represents:

Bit pattern	Effect
ØØ	The double-width pixel takes the background colour and is thus rendered invisible.
Ø1	Sets whatever colour is specified in register V + 37.
1Ø	Sets the 'unique' colour for the multicolour sprite, and displays the colour specified in the sprite colour registers V + 39 to V + 46.
11	This sets the colour specified in register V + 38.

(See REGISTERS for an explanation of the various locations referred to here.)

SPRITE GENERATOR

The following program can be used for defining either multicolour or standard hi-res sprites, storing them in special sprite data files. Up to 64 can be defined for each file. The file can be SAVEd to tape or disk when it is filled in whole or in part. Any number of separate sprite data files may be created—make sure that each is given a unique name.

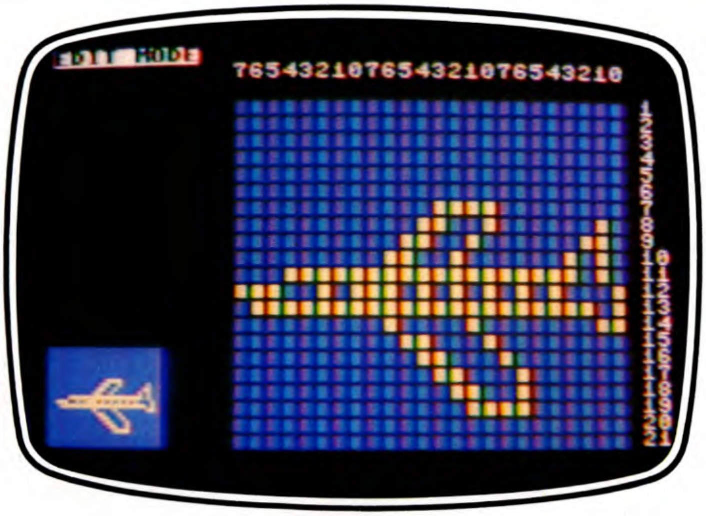
- DEFINING MULTICOLOUR SPRITES
- USING THE SPRITE GENERATOR
- TEMPORARY AND PERMANENT SPRITE STORAGE
- DIRECT DATA ENTRY
- BROWSING THROUGH YOUR SPRITE 'BANK'
- USING DATA FOR YOUR OWN PROGRAMS
- SPRITE CONTROL REGISTERS

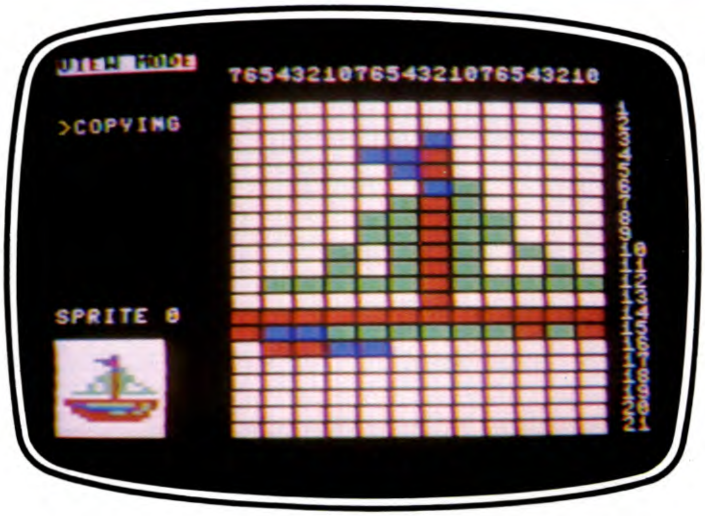
This program creates a sprite storage area which is read from and then written to during the creation of each sprite. You have the option to display any of 64 sprite designs, edit it, or dump the relevant DATA values to the screen or printer where they may be copied for inclusion in your own programs.

The sprite data files are SAVEd as code from locations 12288 to 16384 and may be called directly by your own programs.

Note that some of the end lines contain symbols which are the second letter of the 'shorthand' POKE, and look like pO when the display is toggled into lowercase mode (press SHIFT AND C= key simultaneously). The shorthand method must be used to keep these lines within the 80 character limit. Also note that the symbols which appear in Lines 180 and 190 are obtained by pressing SHIFT plus O, along with C= plus Y in the second line. These lines define the sprite character grid. Remaining symbols refer to 'quote mode' colour changes and cursor controls.

```
10 POKE 51,255:POKE 52,47:POKE
   55,255:POKE 56,47:CLR
20 POKE 53280,6:POKE 53281,6:
   PRINT "♥◣";TAB(12);"□□
   □□□□□□□□□□□";
   CHR$(8)
30 PRINT TAB(12);"▣SPRITE EDITOR▣"
40 CH = 1:PRINT "▤▣▣▣◆DO YOU
   WANT MULTI COLOUR SPRITES.":
   INPUT"(Y/N)";A$
50 IF A$ < > "Y" AND A$ < > "N"
   THEN 40
60 IF A$ = "Y" THEN CH = 2: GOTO 90
70 INPUT "▣ENTER SPRITE COLOUR";
   CL(1):IF CL(1) < 0 OR CL(1) > 15 THEN 70
80 CL(2) = CL(1):CL(3) = CL(1):
   GOTO 130
90 FOR Z = 1 TO 3
100 PRINT "▣ENTER SPRITE COLOUR";Z;
110 INPUT CL(Z):IF CL(Z) < 0 OR CL(Z) > 15
   THEN 100
120 NEXT Z
130 INPUT "▣ENTER BACKGROUND
```





The illustrations show the sprite generator in edit mode for (left) hi-res sprites and (right) multicolour sprites

```
   COLOUR";CL(4): POKE 650,128
140 IF CL(4)<Ø OR CL(4)>15 THEN 13Ø
15Ø IF CL(4)=Ø THEN POKE 5328Ø,11: POKE
   53281,11: GOTO 17Ø
16Ø POKE 5328Ø,Ø:POKE 53281,Ø
17Ø FOR Z=832 TO 894:POKE Z,Ø: NEXT Z
18Ø L=1155:X=Ø:Y=Ø:CC=2Ø7:
   DD=24:G$="◰":C(1)=2Ø7:
   C(2)=2Ø7:C=55427
19Ø IF CH=2 THEN G$="◰◰":
   DD=12:C(2)=247
2ØØ A(1)=128:A(2)=64:A(3)=32:
   A(4)=16:A(5)=8:A(6)=4:
   A(7)=2:A(8)=1
21Ø V=53248:POKE 2Ø4Ø,13: POKE
   V+21,1:POKE V,28:POKE V+1,197:POKE
   V+28,Ø
22Ø POKE V+38,CL(1):POKE
   V+39,CL(2):POKE V+37,CL(3): IF
   CH=2 THEN POKE V+28,1
23Ø POKE V+23,1:POKE V+29,1
24Ø PRINT"♥▣▣▣▣▣▣▣▣
   ▣▣▣▣▣▣▣▣▣": POKE
   646,CL(4)
25Ø FOR Z=1 TO 6:PRINT
   "▨□□□□□□□": NEXT Z
26Ø FF$="":FOR Z=1 TO DD:
   FF$=FF$+G$:NEXT
27Ø PRINT "▤▥▐▐▐▐▐▐▐
   ▐▐▐▐▣7654321Ø7654321Ø
   7654321Ø▣"
28Ø FOR Z=1 TO 21:PRINT
   "▐▐▐▐▐▐▐▐▐▐▐▨
   ";:POKE 646,CL(4)
29Ø PRINT FF$;"▬▥";Z: NEXT Z:POKE
   895,Ø
3ØØ PRINT"▤▨EDIT MODE"
31Ø GET A$:FOR Z=Ø TO CH-1: POKE
   L+X+Y*4Ø+Z, C(Z+1): NEXT Z
32Ø IF A$="E" THEN 137Ø
33Ø IF A$=CHR$(2Ø) OR A$="□" THEN
   AA$=A$:A$="4": GOSUB
   58Ø:A$=AA$
34Ø IF A$="R" THEN POKE V+21,Ø:RUN
35Ø IF A$="*" THEN 122Ø
36Ø IF A$=CHR$(134) THEN POKE V+23,1
37Ø IF A$=CHR$(133) THEN POKE V+29,1
38Ø IF A$=CHR$(136) THEN POKE V+23,Ø
39Ø IF A$=CHR$(135) THEN POKE V+29,Ø
4ØØ IF A$="▌" THEN X=X-CH
41Ø IF A$="◘" THEN Y=Y-1
42Ø IF A$="▐" OR A$="□" THEN
   X=X+CH
43Ø IF(A$="▣" OR A$=CHR$(13)) THEN
   Y=Y+1
44Ø IF X<Ø THEN X=24-CH
45Ø IF X>24-CH THEN X=Ø
46Ø IF Y<Ø THEN Y=2Ø
47Ø IF Y>2Ø THEN Y=Ø
48Ø IF A$=CHR$(13) THEN X=Ø
49Ø IF A$="▤" THEN X=Ø:Y=Ø
5ØØ IF A$="V" THEN GOSUB 1Ø8Ø
51Ø CC=PEEK(L+X+Y*4Ø)
52Ø IF A$="S" THEN GOSUB 77Ø:
   GOSUB 69Ø:GOSUB77Ø:GOTO 3ØØ
53Ø IF A$="C" THEN GOSUB 77Ø:
   GOSUB 78Ø:GOSUB 77Ø:GOTO 3ØØ
54Ø FOR Z=Ø TO CH-1:POKE
   L+X+Y*4Ø+Z,32:NEXT Z
55Ø IF VAL(A$)>Ø AND VAL(A$)<4 THEN
   GOSUB 58Ø
56Ø IF A$="♥" THEN FOR Z=832 TO
   895: POKE Z,Ø:NEXT Z:GOTO 27Ø
57Ø GOTO 31Ø
58Ø XX=832+INT(X/8)+(Y*3):
   PP=PEEK(XX):PE=PEEK(XX) AND
   A(X-(INT(X/8)*8)+1)
59Ø PF=PEEK(XX) AND
   A(X-(INT(X/8)*8)+2)
6ØØ IF CH=2 THEN 64Ø
61Ø IF A$<>"4" AND PE=Ø THEN POKE
   XX,PEEK(XX)+A(X-(INT
   (X/8)*8)+1)
62Ø IF A$="4" AND PE<>Ø THEN POKE
   XX,PEEK(XX)-A(X-(INT
   (X/8)*8)+1)
63Ø GOTO 68Ø
64Ø IF PE=Ø AND(A$="1"ORA$=
   "2")THEN POKE XX,PEEK(XX)+A(X-
   (INT(X/8)*8)+1)
65Ø IF PE<>ØAND(A$="4"ORA$=
   "3")THEN POKEXX,PEEK(XX)-A(X-
   (INT(X/8)*8)+1)
66Ø IF PF=Ø AND(A$="1"ORA$=
   "3")THEN POKE XX,PEEK(XX)+A(X-
   (INT(X/8)*8)+2)
67Ø IF PF<>ØAND(A$="4"ORA$=
   "2")THEN POKEXX,PEEK(XX)-A(X-
   (INT(X/8)*8)+2)
68Ø FOR Z=Ø TO CH-1:POKE C+X+
   Y*4Ø+Z,CL(VAL(A$)):NEXT Z: RETURN
69Ø GOSUB 77Ø:NU=Ø:PRINT
   "▤▨SPRITE NO."
7ØØ NU$="":INPUT "▤▣▣Ø-63";
   NU$:NU=VAL(NU$):IF NU<Ø
   ORNU>63ORLEN(NU$)=Ø THEN 7ØØ
71Ø PRINT "▣▨□□POINTER:":
   PRINT 192+NU
72Ø PRINT "▣▨□START□AD:":
   PRINT 12288+NU*64
73Ø PRINT "▣▨□□□END□AD:":
   PRINT 12288+NU*64+63
74Ø FOR Z=Ø TO 63:POKE 12288+
   NU*64+Z,PEEK(832+Z):NEXT Z
75Ø PRINT "▣>PRESS□KEY":
   PRINT "TO□CONTINUE":POKE 198,Ø:
   WAIT 198,1:POKE 198,Ø
76Ø RETURN
77Ø PRINT"▤";:FOR Z=1 TO 17:
   PRINT"□□□□□□□□□□□":
   NEXT Z:RETURN
78Ø NU$="":PRINT "▤▨SPRITE□
```

```
   NO.":INPUT"▤◘◘Ø–63";
   NU$
79Ø NU=VAL(NU$):IF NU<Ø OR NU>63
   OR LEN(NU$)=Ø THEN 78Ø
8ØØ PRINT "▤◘◘◘◘>
   COPYING"
81Ø NU=12288+NU*64:PO=832:
   IF CH=2 THEN 87Ø
82Ø TT=Ø:FOR Z1=Ø TO 2Ø:FOR Z2=Ø TO
   2:FOR Z3=Ø TO 7
83Ø XX=C+(Z1*4Ø)+(Z2*8)+Z3
84Ø IF(PEEK(NU+TT)ANDA(Z3+1))
   <>ØTHEN POKE XX,CL(2):GOTO 86Ø
85Ø POKE XX,CL(4)
86Ø NEXT Z3:POKE 832+TT,PEEK(NU+
   TT):TT=TT+1:NEXT Z2,Z1:RETURN
87Ø TT=Ø:FOR Z1=Ø TO 2Ø:FOR Z2=Ø TO
   2:FOR Z3=Ø TO 7 STEP 2
88Ø XX=C+(Z1*4Ø)+(Z2*8)+Z3
89Ø R1=(PEEK(NU+TT)ANDA(Z3+1))
9ØØ R2=(PEEK(NU+TT)ANDA(Z3+2))
91Ø IF R1<>Ø AND R2<>Ø THEN POKE
   XX,CL(1):POKE XX+1,CL(1)
92Ø IF R1<>Ø AND R2=Ø THEN POKE
   XX,CL(2):POKE XX+1,CL(2)
93Ø IF R1=Ø AND R2<>Ø THEN POKE
   XX,CL(3):POKE XX+1,CL(3)
94Ø IF R1=Ø AND R2=Ø THEN POKE
   XX,CL(4):POKE XX+1,CL(4)
95Ø NEXT Z3:POKE 832+TT,PEEK(NU+
   TT):TT=TT+1:NEXT Z2,Z1:RETURN
96Ø I$="":INPUT "♡(P)RINTER OR
   (S)CREEN";I$
97Ø IF I$<>"P" AND I$<>"S" THEN
   96Ø
98Ø IF I$="P" THEN OPEN 4,4:CMD 4
99Ø PRINT "◘SPRITE DATA";LU:LN=
   (192+LU)*64:PRINT
1ØØØ FOR Z=LN TO LN+62:PRINT LEFT$
   (STR$(PEEK(Z))+"□□□",4);:
   NEXT Z
1Ø1Ø PRINT "▐Ø":PRINT "◘SPRITE
   POINTER=";192+LU
1Ø2Ø PRINT "◘START□ADDRESS
   □=";LN
1Ø3Ø PRINT "◘END□ADDRESS
   □□□=";LN+63
1Ø4Ø IF I$="S" THEN PRINT
   "◘>PRESS□ANY□KEY□FOR
   □EDIT□MODE"
1Ø5Ø IF I$="S" THEN POKE 198,Ø:
   WAIT 198,1:POKE 198,Ø
1Ø6Ø IF I$="P" THEN PRINT#4:CLOSE 4
1Ø7Ø GOTO 17Ø
1Ø8Ø PRINT"▤◨VIEW□MODE":
   FOR Z=Ø TO 63
1Ø9Ø NU=12288+Z*64
11ØØ PRINT"▤◘◘◘◘◘◘◘◘
   ◘◘◘◘◘◘◘◘▥
   SPRITE□□□□▮▮▮▮▮▮▮▮";Z
111Ø POKE 2Ø4Ø,192+Z
112Ø GET R$
113Ø IF R$=CHR$(13) THEN 121Ø
114Ø IF R$="B" THEN Z=Z–1:
   IF Z<>–1 THEN 1Ø9Ø
115Ø IF Z=–1 THEN Z=63:GOTO 1Ø9Ø
116Ø IF R$="C" THEN NU=Z:
   GOSUB 8ØØ: GOTO 121Ø
117Ø IF R$="□" THEN 12ØØ
118Ø IF R$="D" THEN POKE V+21,Ø:
   LU=Z:GOTO 96Ø
119Ø GOTO 112Ø
12ØØ NEXT Z:GOTO 1Ø8Ø
121Ø POKE 2Ø4Ø,13:GOSUB 77Ø:
   GOTO 3ØØ
122Ø GOSUB 77Ø:PRINT "▤◘▐▐▐
   DO□YOU□WANT□TO□STORE□
   SPRITE□(Y/N)?"
123Ø GET R$:IFR$="N" THEN 127Ø
124Ø IF R$="Y" THEN 126Ø
125Ø GOTO 123Ø
126Ø PRINT"▤◘";:FOR Z=1 TO 39:
   PRINT "□";:NEXT Z:GOSUB 69Ø
127Ø POKE V+21,Ø:POKE 5328Ø,14:POKE
   53281,6
128Ø PRINT "♡◆DO□YOU□WANT
   □TO□SAVE□OR□LOAD□DATA□
   (S/L)?":S$="LOAD"
129Ø GET F$:IF F$="S" THEN S$=
   "SAVE":GOTO 131Ø
13ØØ IF F$<>"L" THEN 129Ø
131Ø PRINT "♡◨";TAB(13);S$;
   "□ROUTINE"
132Ø IFF$="L"THENPRINT"◘LOAD"
   CHR$(34)"SPRITE□FILE□1";
   CHR$(34);",1,1":GOTO135Ø
133Ø PRINT "◘P□43,Ø:P□44,48:
   P□45,Ø:P□46,64:";
134Ø PRINT "SAVE";CHR$(34);"SPRITE
   □FILE□1";CHR$(34);",1,1"
135Ø PRINT"◘◘◘◘◘◘◘◘◘
   ◘◘P□43,1:P□44,8:P□45,";
   PEEK(45);":P□46,";PEEK(46);
   ":RUN"
136Ø END
137Ø POKE V+21,Ø:PRINT "♡◨
   ENTER□SPRITE□NUMBER,
   FOLLOWED□BY□DATA"
138Ø NU$="":INPUT "▤◘◘Ø–63";
   NU$:NU=VAL(NU$):IFNU<ØORNU>
   63ORLEN(NU$)=ØTHEN137Ø
139Ø FOR Z=Ø TO 63
14ØØ PRINT "♡DATA";Z;:X=Ø:
   INPUT X:IF X<Ø OR
   X>255 THEN 14ØØ
141Ø POKE(192+NU)*64+Z,X:NEXT Z
142Ø PRINT "♡DO□YOU□WANT□
   TO□ENTER□MORE□
   DATA□(Y/N)?"
143Ø GET A$:IF A$="N" THEN 17Ø
144Ø IF A$="Y" THEN 137Ø
145Ø GOTO 143Ø
```

STARTING OFF

When you RUN the program, the first prompt asks you if you want multicolour sprites. If you respond with Y RETURN for yes, the first of three prompts requesting colour selection appears. The colour value you enter can be in the range Ø to 15, following the conventional Commodore 64 scale starting with Ø as black. Three colours can be selected—the fourth colour chosen is for background colour.

If you press N and RETURN, the hi-res sprite mode is selected in which you get only two colour prompts. Otherwise the remaining routines are the same.

When you've selected your colours, the screen displays the sprite generation grid, a status or information panel on its left, and a sprite display area a little lower down. The grid has a resolution of 12 horizontal squares in multicolour mode, 24 in hi-res.

EDITING

Once you've made your initial selection of colours, the program automatically enters 'edit' mode and you can begin entering a pattern within the grid. Use the cursor keys to move around, and keys 1, 2 and 3 to colour a square.

There are several other editing features. Pressing RETURN takes the cursor to the start of the next line, CLR/HOME on its own homes the cursor, SHIFT plus CLR/HOME clears the screen, SPACE clears the preceding square, INST/DEL clears squares under the cursor.

The sprite display area shows plotting or erasing as it takes place, and at any stage actually shows what the sprite looks like. You can compare its form in normal and expanded modes by making use of the function keys. Pressing f1 expands the sprite horizontally, f3 vertically. f5 and f7 return the sprite to normal size in X and Y axes respectively. The program starts off with an expanded sprite.

If at any point during the edit mode you feel that the choice of sprite type or colours needs changing, press R to initiate a 'restart'.

SPRITE STORAGE

A newly defined or amended sprite can be returned to storage from its temporary location in the cassette buffer (locations 832 upwards are used) by pressing S at any time.

To amend an existing sprite, perhaps in the course of using it as the basis of another sprite design, press C. This puts the program into 'copy' mode. You are then prompted for a sprite number. Any number between Ø and 63 is acceptable. The sprite you choose is then copied from its relevant place in memory and shown both on the grid and in the sprite display area.

When you first use the program, all the sprite memory locations contain garbage, some of which makes for interesting sprites! Try selecting a sprite number, 56, for example, and see what you get.

The sprite that is displayed following the copy mode can be worked on or returned to memory, as it can at any future point, simply by pressing S, the 'storage' key. You are then again asked to enter a number in the range Ø to 63. If you want to overwrite the sprite that you used as the starting point of the new design, simply enter the same number that you used to call it up. Otherwise, choose another number, but note that the sprite which originally occupied the storage location you pick is overwritten in the process.

When the sprite is consigned to memory, additional information is displayed to the left of the grid. Details of that particular sprite pointer, start address and finish address can be noted for future use or printed out for a hard copy (as explained below).

BROWSING

You can browse through what's in sprite memory at any time during edit mode by pressing V for 'view'. The sprite on which you were working remains in memory until overwritten by a storage instruction.

The display starts at sprite Ø and can be advanced very quickly through the entire range, one sprite at a time, by pressing the space bar. The ease with which this can be done illustrates the power of simply altering the *sprite pointers*. To go backwards one sprite place, press B. If you want to use two similar sprites as the basis of an animation, you can store them in two adjacent locations and toggle back and forwards between them in this way to check the effect.

If you find a sprite you wish to revise, or use as the basis of another one, use the copy routine, key C. Otherwise press RETURN to enter edit mode again.

PRINT DATA

If you have a printer connected and switched on, you can obtain a hard copy of the sprite number, pointer, start and end addresses by pressing D, for 'data', in view mode. You are offered output to a screen or printer.

The data can then be transferred to other programs by manual entry.

DATA ENTRY

Sprite definitions in the form of decimal, DATA statements are frequently used in listings and may also be entered directly into the sprite generation program. Thus you could use published DATA values as a basis for your own designs. Some examples are shown in this article—and you should transfer these to memory using the procedure outlined in the next paragraph.

When you are in edit mode, press E to make a data 'entry'. You are first prompted for the sprite number you wish to assign to the new design, then for the 64 separate items of data. When you've completed the entry, you are asked whether or not you wish to continue. If not, the program returns you to edit mode and a display of the last sprite entered, which is also stored in memory.

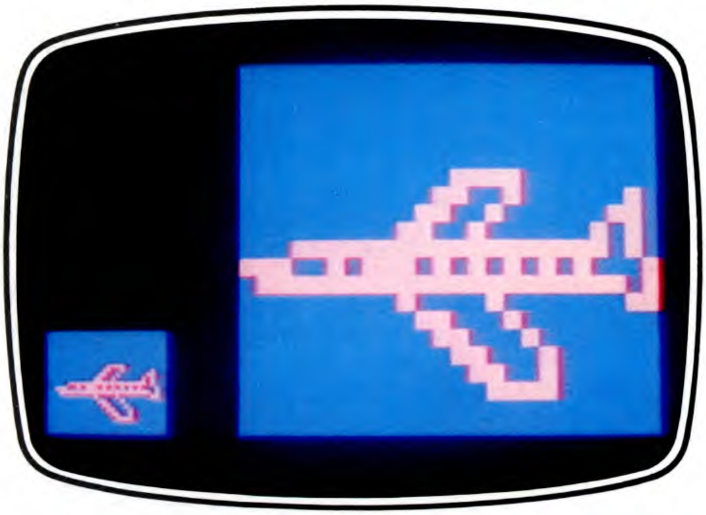
SAVEING SPRITE DATA

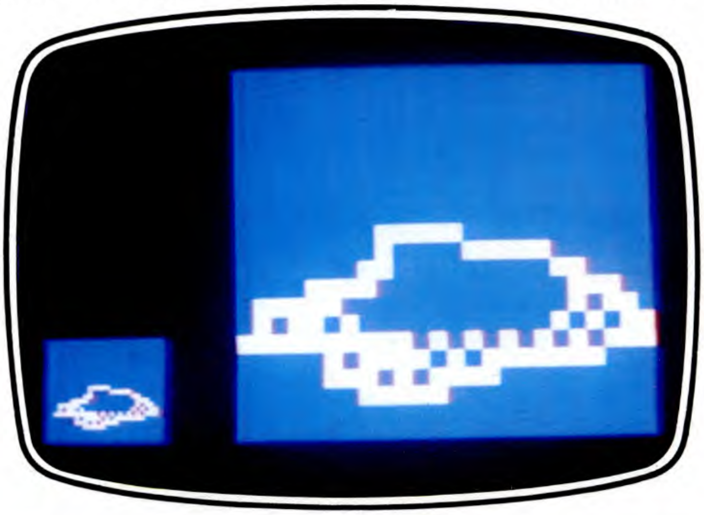
The sprite data held in memory can be SAVEd at any time. To do this, press * to exit the edit mode. You are asked whether or not you want to SAVE the last sprite you were working on. This is a precautionary measure: press either Y or N. You are returned to edit mode (where you can use the store function) or asked whether you want to SAVE or to LOAD. If you select SAVE, the program presents you with a screen showing preprinted direct commands.

To use the SAVE routine, simply move the cursor up to the first of the two lines, enter a suitable sprite file name and press RETURN. The line is actually set up for tape users: if you wish to save the disk, alter the *device number* in Line 134Ø to 8. This is the second to last number in that line. The alteration can, of course, be made in direct mode.

To abort the SAVE option or return to the

Each of these sprites can be entered directly into the sprite generator program using the data entry option. The appropriate data is shown below each illustration on these pages





```
Ø   Ø   Ø   Ø   Ø   Ø   Ø   Ø   Ø   Ø   Ø   Ø   Ø   Ø   Ø   Ø   Ø   Ø   Ø   15
Ø   Ø   25  2   Ø   50  6   Ø   100 1Ø  31  255 25Ø 229 213 81  127 255 253 Ø
66  3   Ø   49  Ø   Ø   24  128 Ø   12  64  Ø   6   64  Ø   3   192 Ø   Ø   Ø
Ø   Ø   Ø   Ø
sprite pointer=193
start address =12352
end address  =12415
```

```
Ø   Ø   Ø   Ø   Ø   Ø   Ø   Ø   Ø   Ø   Ø   Ø   Ø   Ø   Ø   Ø   Ø   Ø   Ø   Ø
Ø   Ø   Ø   Ø   Ø   Ø   Ø   Ø   248 Ø   Ø   135 192 1   128 48  15  Ø   28  124
Ø   22  9Ø  Ø   43  255 213 87  5   235 248 7   94  Ø   1   24Ø Ø   Ø   Ø   Ø
Ø   Ø   Ø   Ø
sprite pointer=196
start address =12544
end address  =126Ø7
```

program following a SAVE, simply run the cursor down to the lower line and press RETURN.

Note that the entire section of memory between 12288 and 16384 is SAVEd, on the assumption that data for all 64 sprites is wanted. This may not, of course, be the case, and you can SAVE a smaller section of memory by adjusting the POKEd value in location 46. It is best to make any amendment of this type in direct mode to the first line of the save routine display. All you have to do is simply change the value 64 to whatever number of sprites you want to SAVE.

LOADING SPRITE DATA

You can LOAD sprite data in any one of several ways. From within the program, follow the same routine for SAVEing up to the point where the LOAD decision is made. The loading routine screen is similar to that of the save routine—simply run the cursor up to the first line, enter the name of the sprite file you wish to LOAD and press RETURN. You can change the device number either in direct mode or more lastingly (Line 132Ø) as before.

When the file has been LOADed, move the cursor to the second of the preprinted direct commands and press RETURN. This returns you to the program, with the new sprite data in memory. Make sure that the original sprite data has been SAVEd beforehand if it is wanted, as this area of memory is overwritten by the new information.

YOUR OWN PROGRAMS

The sprite file data can also be used by your own programs. Use the same LOADing procedure as before but follow this with a NEW. Then LOAD your own program, which must be smaller than 12k to avoid memory conflict with the sprite storage area.

If you wish to use a sprite data file without having to use the main sprite generation program, type:

```
LOAD "SPRITE FILE NAME",1,1
```

Remember to use exactly the right name for the sprite file, and to change the first number from 1 to 8 if you're using a disk unit. The sprite data is then LOADed into its original location.

Once in memory, the sprite data must be protected from subsequent program LOADs, so enter this as a single direct command sequence:

```
POKE 51,255: POKE 52,47: POKE 55,255:
POKE 56,47: CLR
```

You may find it better to make this the first line of your program.

Your program of course has to access specific parts of the storage area and this is where hardcopy of the pointer and address data is invaluable. You have to define the colours again to match your original selection. For a multicolour sprite, the values (Ø–15) of the original colours c1, c2 and c3 are POKEd into the following locations: (n is the sprite number Ø–7). V must first be set as 53248, the start of the registers of the VIC-II chip (see below).

```
POKE V+38, c1
POKE V+39+n, c2
POKE V+37, c3
```

For hi-res sprites only the single colour location V+39+n is needed. But note that the program must set up multicolour mode first. This means that Bit 4 of location must be set high—achieved by a solo POKE of 16 in location V+22.

REGISTERS

Much the hardest part of using sprites is setting up all the various registers needed to control their movement, colour, position and priority.

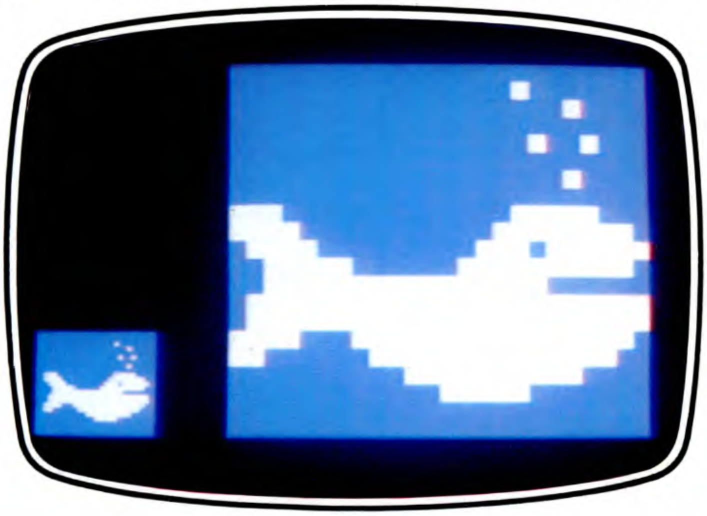
The 47 registers of the VIC-II chip control all of the functions to do with sprites. These are listed in table form on page 172. Let's take a closer look at some of these:

SPRITE POSITIONING

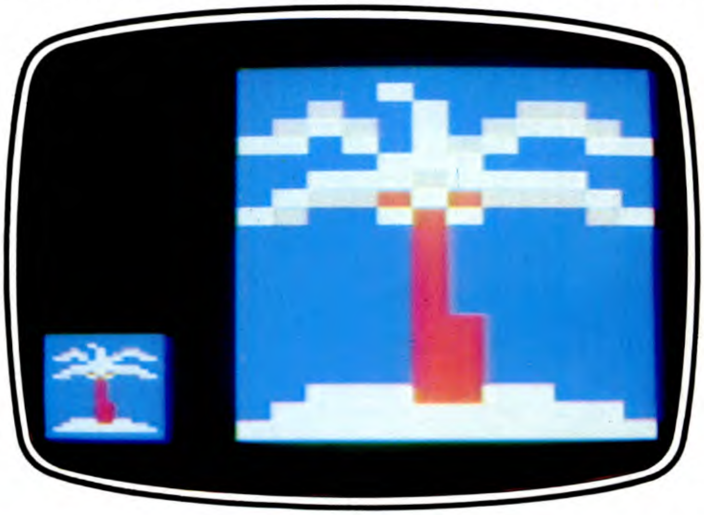
V+Ø and V+1 These two registers determine the X and Y coordinates of spite Ø. Two POKEs are needed to position a sprite:

```
POKE V+Ø,horizontal pixel position
POKE V+1,vertical pixel position
```

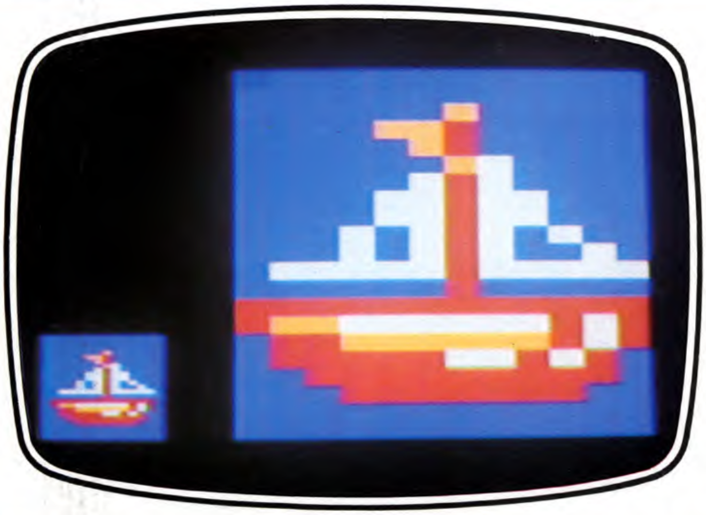
The pixel position can be in the range Ø to 32Ø horizontally, Ø to 2ØØ vertically (but only Ø to

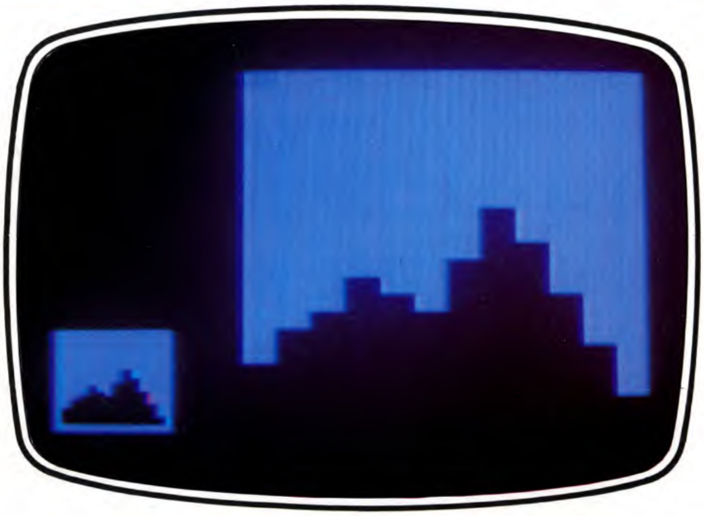


```
Ø   Ø   Ø   Ø   Ø   128 Ø   Ø   16  Ø   Ø   Ø   Ø   Ø   72  Ø   Ø   Ø   Ø   Ø
16  Ø   Ø   Ø   224 Ø   248 24Ø 1   254 12Ø 3   191 62  7   254 63  255 192 127
255 255 127 255 255 241 255 254 192 255 252 Ø   63  24Ø Ø   15  128 Ø   Ø   Ø
Ø   Ø   Ø   Ø
sprite pointer=197
start address =126Ø8
end address  =12671
```



```
Ø   Ø   Ø   Ø   64. Ø   8   16  16Ø 38  18  88  81  25  5   Ø   84  Ø   5   1Ø1
8Ø  26  222 164 96  116 9   Ø   48  Ø   Ø   48  Ø   Ø   48  Ø   Ø   48  Ø   Ø
48  Ø   Ø   6Ø  Ø   Ø   6Ø  Ø   Ø   6Ø  Ø   Ø   6Ø  Ø   5   125 64  21  85  84
85  85  85  Ø
sprite pointer=194
start address =12416
end address  =12479
```





```
0   0   0   0   0   0   0   8   0   0   172 0   0   44  0   0   9   0   0   29
0   0   93  64  0   93  64  1   29  16  1   29  4   21  93  85  0   12  0   255
255 255 233 85  119 62  170 183 63  245 220 15  255 252 3   255 240 0   0   0
0   0   0   0
sprite pointer=192
start address =12288
end address  =12351
```

```
0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   2   0   0   2   0   0   2   128 0   10  128 2   10  128 2
138 160 10  170 160 42  170 160 42  170 168 170 170 168 170 170 168 170 170 170
170 170 170 0
sprite pointer=199
start address 12736
end address 12799
```

160 horizontally in multicolour mode).

Each register can hold a maximum value of 255 when all bits are 'on'—so there's a very obvious problem when it comes to positioning a sprite at a location towards the right extreme of the screen.

There is a simple solution, however, and this is provided by another location, V+16, which provides the MSB (most significant bit) value on those occasions when it is required for large X values, those above 255.

Each bit of this register looks after one of the sprites: thus you would set bit 0 to 'on' for sprite 0, bit 1 to 'on' for sprite 1, and so on.

Each is a high order bit which adds 255 to the addressing. Thus POKE V+16,1 followed by POKE V+0,10 would position sprite 0 at location 265. To return to positions less than 255, location V+16 has to be zeroed using POKE V+16,0.

If several sprites have to access the V+16 bits, clearly some have to be set high while others are low. POKEing V+16 with values in the range 1 to 255 can selectively turn on from one to all of the sprites. Logical operators AND and OR or NOT can also be used to selectively control the bits in a situation like this, particularly when it comes to switching out the V+16 effect.

SPRITE MOVEMENT

Because they control positioning, registers V+0 and V+1 come into use for programming movement of sprite 0—all you have to do is set up a suitable FOR ... NEXT loop to move sprite 0 one horizontal or vertical position, at a time. Another loop may be used to combine movement in the other direction. Still another loop may be used for controlling the speed of movement, by introducing a delay in one or other of the movement loops. **V+2 to V+15** These are the X and Y registers for the remaining seven sprites, which are treated in exactly the same way as V+0 and V+1.

VIC CONTROL

V+17 and V+22 These are the two VIC-II chip control registers. Between them they control the primary settings of the VIC chip—things like the number of display columns and rows, the colour mode in use, and reset.

Each of these registers relies on bit control. Bits 0, 1 and 2 of V+17 regulate the number of pixels for Y-axis (vertical) smooth scrolling. Bit 3 sets the number of display rows—set low results in 24, set high gives 25. Bit 4 gives a blank screen when set low—the screen and border colour are matched. Bit 5 and Bit 6 give standard and extended bit-map modes, respectively, when set high. Bit 7 is the high-order bit value for V+18 (see below), used when values in excess of 255 are required (something like the way the V+16 bits are used for sprite positioning).

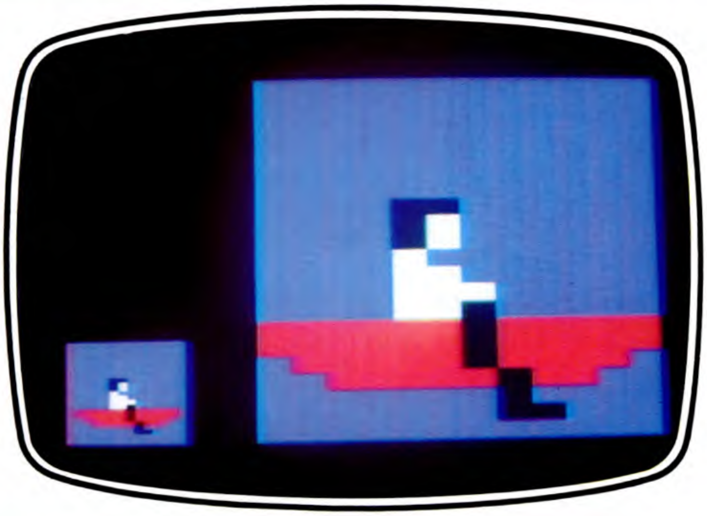
The bits of control register two (V+22) exercise similar control. Bits 0, 1 and 2 designate the number of pixels for X-axis (horizontal) smooth scrolling. Bit 3 sets 38 column width when set low, 40 columns when set high. Bit 4 activates multicolour mode when set high and allows text as well as bit-mapped graphics. Bit 5 resets the VIC chip when set high. Bits 6 and 7 remain unused.

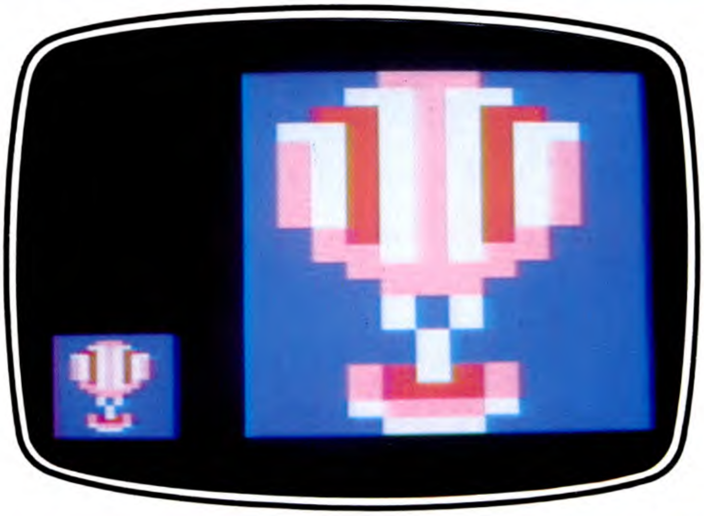
V+18 This is the location of the register which, when read, contains the current position of the raster (a definition of which is the set of scanning lines which appear as a patch of light by which the TV image is produced). When written to, a *raster interrupt* is triggered when the raster value matches what's in the register. This register is an important one if more than the normal number of sprites are to be displayed on the screen, a subject covered in a later issue.

SPRITE ENABLE

V+21 This the register which does all the work of switching sprites both off and on. Each bit controls one sprite, in the usual order—Bit 0 for sprite 0, Bit 1 for sprite 1, and so on. A sprite is enabled (switched on) by setting the control bit high. By judicious use of POKE values, and AND and OR or NOT masking, very selective control can be exercised over the switching process.

All sprites can be switched off by POKEing zero into this location—do this towards the end of a sprite display sequence.





```
0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   160 0   0   144 0   0   144 0   0   64  0   0   80  0   0   84  0   0
88  0   255 251 255 255 251 255 63  251 252 15  254 240 0   2   0   0   2   128
0   0   0   0
sprite pointer = 201
start address = 12864
end address  = 12927
```

```
0   168 0   1   101 0   15  103 192 23  103 80  39  103 96  39  103 96  39  103
96  39  103 96  39  103 96  11  103 128 10  102 128 2   170 0   0   168 0   0
68  0   0   68  0   0   16  0   0   16  0   2   222 0   2   254 0   1   169 0
0   84  0   0
sprite pointer = 195
start address = 12480
end address  = 12543
```

EXPANSION

V + 23 and V + 29 One of the special features of sprites is their ability to expand instantaneously in vertical and/or horizontal planes to four times their original size. These are the two locations (X and Y respectively) which handle this job.

Once again, each of the eight sprites is controlled by the appropriate bit. Bit Ø in V + 23 and V + 29 looks after the Y and X expansion of sprite Ø, Bit 1 does the same for sprite 1 ... and so on. Simply set the appropriate bit(s) high, again using the logical operators for selective control over the eight separate sprites.

PRIORITY

V + 27 Some interesting effects are possible by making sprites pass over one another, and in front of or behind the background (such as a text or ROM graphic character). The latter function is controlled by this register.

Sprites take their *priority* from the sprite number—the lower the number, the greater the priority. Thus sprite 5 has greater priority than sprite 7, and will pass over it (in front of it) if both are displayed together and set for path which in part takes them over the same group of screen coordinates. Sprite Ø has priority over all other sprites.

The sprite-background priority is established for each sprite by standard bit control of V + 27. These are normally set to zero (that is, low or off) but can be set high using direct POKE values. Selective control is again possible using the logical operators. Set high (using POKE V+27,4) Bit 3 would give the background higher priority than sprite 3.

COLLISIONS

V + 30 and V + 31 These two locations are used to detect 'collisions' between sprite and sprite or sprite and background, respectively.

Collision detection plays an immensely important part in games programming but has uses in other fields too if sprite overlapping needs to be detected. The contents of the two registers remains zero until two or more sprites collide or the registers are reset.

By PEEKing V + 3Ø it is possible to tell which sprites have been in collision. If the value returned is, for example, 129, this correspond with the setting of Bits 7 and Ø—indicating that sprites Ø and 7 had collided.

In a similar fashion the bit value obtained by PEEKing V + 31 reveals which sprite(s) have collided with the background. Note that the bits remain set even after they've been read and must be cleared before any further sequence of collisions has to be detected. Use is also made of registers V + 25 and V + 26 where selective bits are set until read.

COLOURS

V + 32 and V + 33 These are the two familiar locations for changing border and screen (background Ø) colours—perhaps more easily identifiable as locations 5328Ø and 53281 frequently used at the start of programs to avoid the normal default light-blue/blue display colour combination.

V + 34 to V + 36 These are three other locations for contolling background colours 1 to 3 and are used for multicolour character generation where characters may be displayed in one of four colours. The mode is specified in V + 22 and the colour is taken from one of these registers. Only the first four bits are used as the colour values encompass the range Ø to 15 and no greater.

V + 37 and V + 38 These are the sprite multicolour registers. They are used in conjunction with V + 28, which is the register for setting multicolour mode (again with one bit looking after one sprite). To set a sprite to multicolour mode you have to set the appropriate bits of V + 28, V + 17 and V + 22. Part of the colour information for a multicolour sprite comes from the screen colour location, V + 39 to 46 (see below) and V + 37 and V + 38. Each multicolour sprite can have its own colour and two shared colours.

V + 39 to V + 46 These eight locations contain the colour information for sprites Ø to 7. In multicolour mode they provide the 'unique' colour in any particular configuration. Some examples of using these locations are found actually within the sprite generator program.

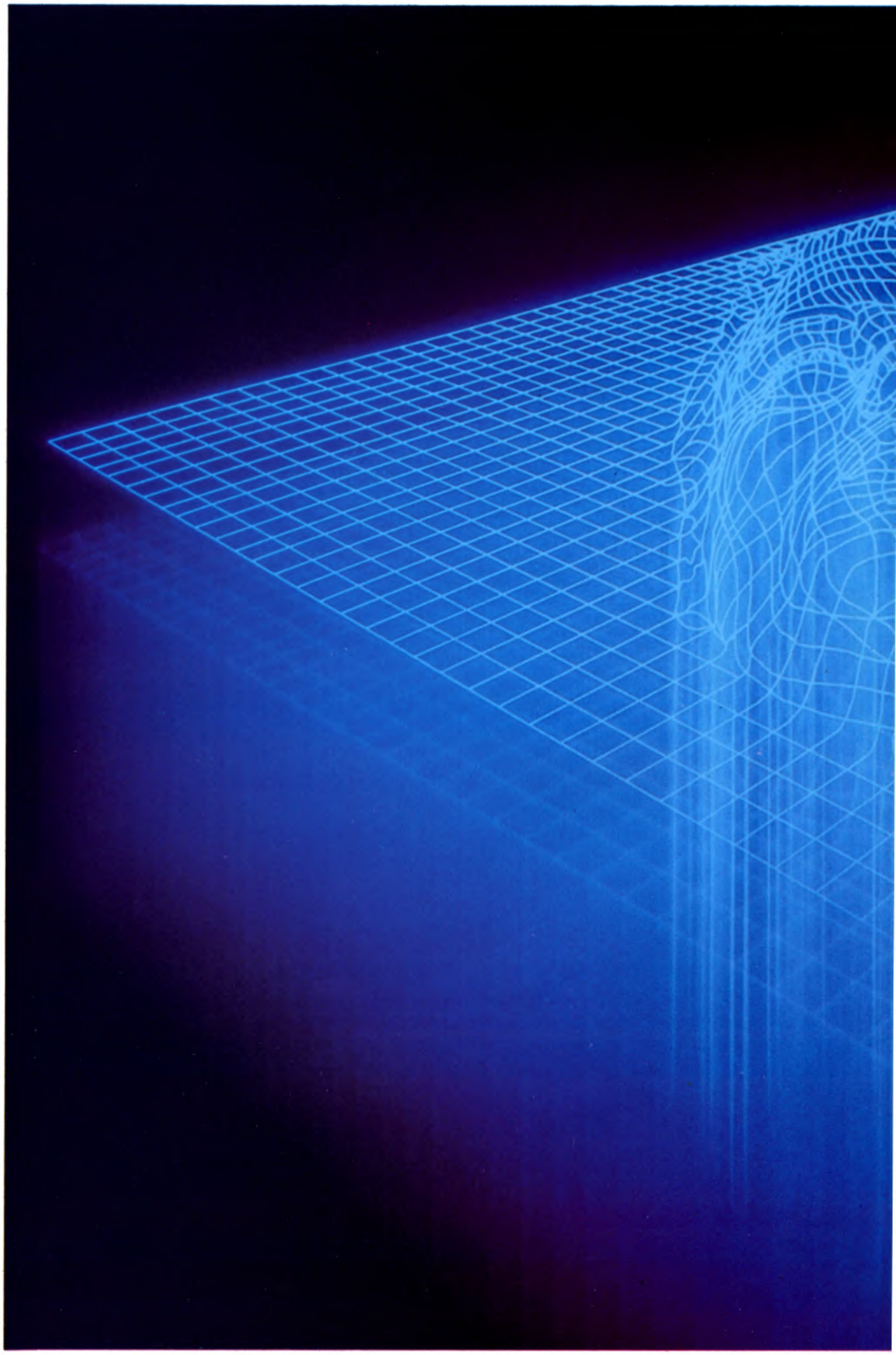
UNDERSTANDING 'FRAMEPRINT'

By now you should know enough about machine code to understand how the frame-print routine given in part one of *INPUT* works, so here it is disassembled

In *INPUT*'s article on machine code on pages 8 to 15, Spectrum, Acorn and Dragon and Tandy owners were told to enter some DATA which would create a frame that could then be used to create large, fast-moving graphics characters or UDGs. It was, in fact, a small machine code program. At that time you were told to enter the figures and not to worry about what they meant.

By now you do know a bit about machine code though. And it is time that you did understand what the numbers meant. So the machine code has been *disassembled* to turn the numbers back into assembly language mnemonics. That way it will be easy to see what is going on.

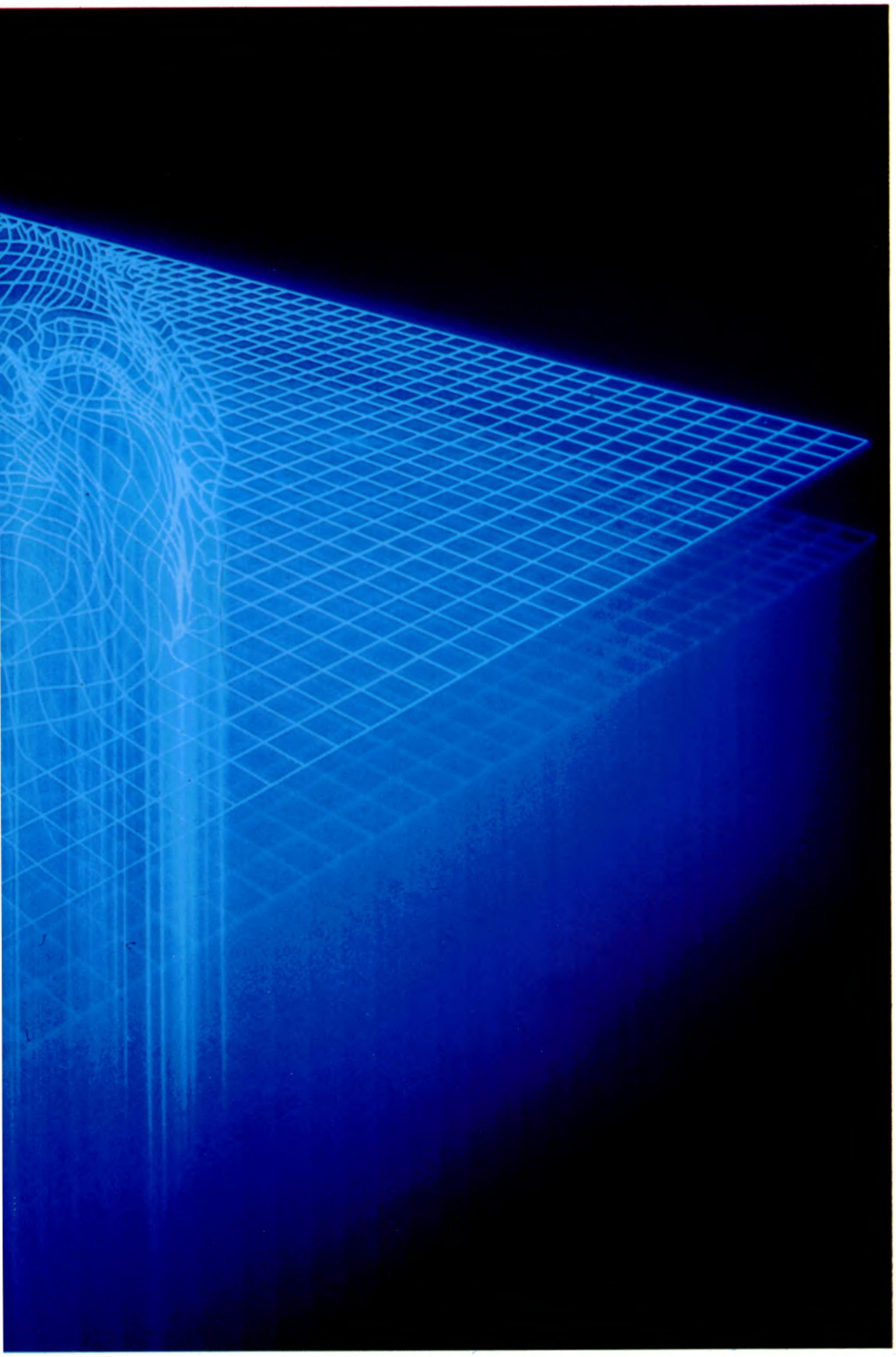
Commodore users do not need to worry about this, because the Commodore program used sprites to create the graphics characters and so did not need a machine code routine.



```
org   65200 (org 32400 on 16K Spectrum)
      jr start
mode  defb 1
data  defb 22
      defb Ø
      defb Ø
      defb 32
      defb 32
      defb 32
      defb 22
      defb Ø
      defb Ø
      defb 32
      defb 32
      defb 32
      defb 22
      defb Ø
      defb Ø
      defb 32
      defb 32
      defb 32
      defb 22
      defb Ø
      defb Ø
      defb 144
      defb 145
      defb 146
      defb 22
      defb Ø
```

```
      defb Ø
      defb 147
      defb 148
      defb 149
      defb 22
      defb Ø
      defb Ø
      defb 15Ø
      defb 151
      defb 152
      defb 22
      defb Ø
      defb Ø
      defb 153
      defb 154
      defb 155
      defb 22
      defb Ø
      defb Ø
      defb 156
      defb 157
      defb 158
      defb 22
      defb Ø
      defb Ø
      defb 159
      defb 16Ø
      defb 161
```

- PRINTING AND UNPRINTING
- CALCULATING PRINT POSITIONS
- MOVING FROM UDG TO UDG
- OPENING CHANNELS
- REFERRING TO TABLES
- MOVING POINTERS
- COMMANDING THE CURSOR
- SETTING UP COUNTERS
- SCREEN PRINT ROUTINES



```
start  ld a,(mode)
       cp 1
       ld bc,18
       jr z,print
       jr c,reset
       sla c
       jr print
reset  ld c,Ø
print  ld ix,data
       add ix,bc
       ld a,(23689)
       ld b,a
       ld a,24
       sub b
       ld (ix+1),a
       inc a
       ld (ix+7),a
       inc a
       ld (ix+13),a
       ld a,(23688)
       ld b,a
       ld a,33
       sub b
       ld (ix+2),a
       ld (ix+8),a
       ld (ix+14),a
       push ix
       ld a,2
       call $16Ø1
       pop de
       ld bc,18
       call $2Ø3C
       ret
```

The first instruction jumps the processor over the long data table which follows it, onto the beginning of the program proper. You could, of course, simply call the program at the first instruction past the data table. But you will find it easier to use the origin address if you want to call the program.

The first byte of data—in fact, the one after the **mode** label—tells the program which frame to print. The BASIC POKEs Ø, 1 or 2 into this location. In this case, the Ø UDG is an empty frame used to clear the screen and 1 and 2 are actually the UDGs—in the case of the tank, they were the two tanks, one pointing in one direction, the other in the other. Here, if no frame number is specified, the routine will default to 1.

THE UDG DATA

After the label **data**, comes the details of three grids. The Spectrum prints its UDGs out in the form of strings. But first it needs to be told where to print them. The instructions for this are contained in the string data too—22 is the token for the BASIC AT and the two zeros are empty bytes. The routine is going to fill them in later with the print position.

The three sets of three 32s make up the empty 9 × 9 frame. 32 is the ASCII code for a space. Each set of three makes up one line of the frame and must be prefaced with its own AT and print position. This is frame Ø.

Frame 1 is made up of UDGs 144 to 152.

Again, in data, these come in sets of three with an AT and two free bytes for the print position in front. And frame 2 is made up of UDGs 153 to 161.

WHICH UDG?

The **ld a,(mode)** loads the number POKEd into the **mode** byte into the accumulator. And **cp 1** compares it with 1.

The BC register pair are then loaded with 18. In fact, the 18 goes into the C register and B is cleared. The only thing that counts during the main routine is the value of the C register, but it will need the B register clear later, when ROM routines are **call**ed.

If the mode number loaded into the accumulator was 1, **cp 1** would have set the zero flag. So **jr z,print** jumps down to the routine labelled **print**. There, the IX register is loaded with the address of the first byte of the UDG data and the contents of the BC register—18—is added to it. Each frame contains eighteen bytes of data—nine for UDGs, three for the ATs and six for the print positions. So this effectively steps the IX register over frame Ø to the beginning of frame 1.

If the mode number was Ø, though, **cp 1** would set the carry flag. So **jr c,reset** jumps to the **reset** label and loads C with Ø. Then **add ix,bc** adds to the address of the **data** label, leaving the IX point at the beginning of the empty frame.

If the mode number is neither 1 nor Ø, it must be 2. So the processor goes onto the instruction **sla c**. This means **s**hift **l**eft **a**rithmetic and acts on the C register. It's called an arithmetic shift because, by shifting all the bits one place to the left, it effectively multiplies the contents of the register by two.

In this case, it doubles the 18 to 36, then jumps to the **print** label. Then when BC is added to the IX pointer, it is shifted 36 bytes along the data table to a position at the beginning ot frame 2.

THE PRINT POSITIONS

The system variable in 23,689 contains the vertical print position. And the print position specified by the BASIC program is that of the top of the frame. Unfortunately, this system variable counts from 1 at the bottom of the screen to 24 at the top, rather than the other way round as in BASIC. So it has to be transferred into the B register, then 24 is loaded into A and the value in B is subtracted from it.

IX contains the address of the first byte of the frame that the routine is going to print. So **ld (ix+1),a** loads the vertical print position of the top of the frame into the next byte down the table—the first Ø that has been left empty.

A is then incremented to give the vertical print position of the next line of UDGs down the frame. This is loaded into the eighth byte down the table by **ld (ix+7),a**. Then A is incremented again for the third line, and this is loaded into the fourteenth byte.

This operation has loaded the vertical print position into the empty bytes immediately following the 22 AT token.

The horizontal print position is then loaded into the accumulator from the system variable 23,688. Again this works back to front compared with BASIC. So it is loaded into B, A is loaded with 33 and B is taken away from A to give the horizontal print position the right way round.

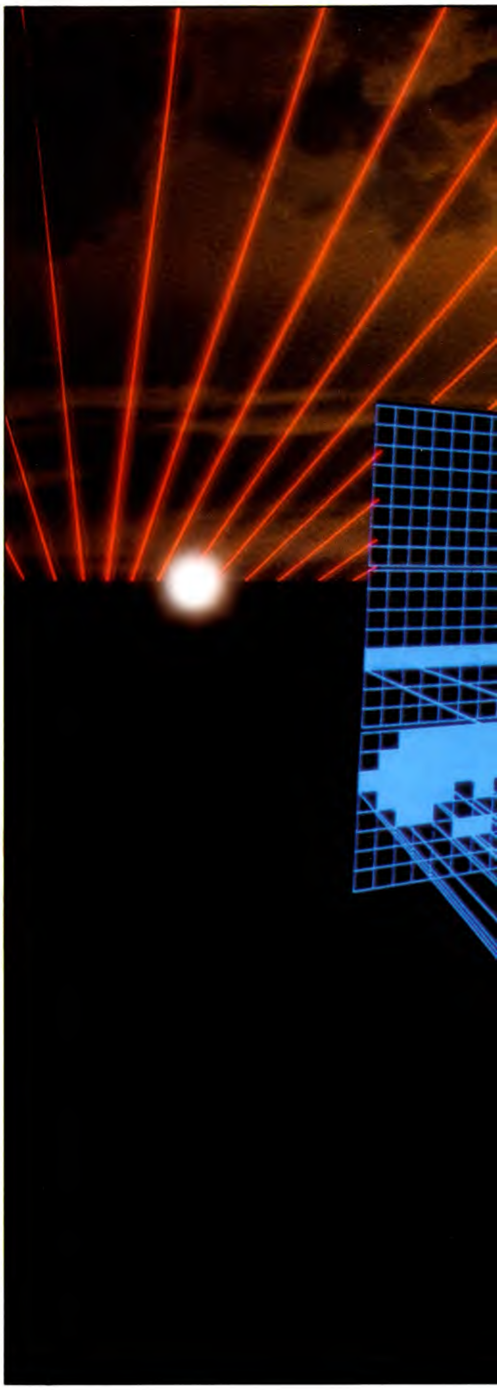
As each line starts at the same horizontal position, it means that the same value of A can be loaded into the other empty spaces in the data table. The **ld (ix+2),a, ld (ix+8),a** and **ld (ix+14),a** do that.

The IX pointer is then **push**ed onto the stack to protect it against being corrupted by the ROM routine that is about to be **call**ed. Then A is loaded with 2 and the channel open routine at 1,6Ø1 is **call**ed. The 2 in the accumulator defines the channel to be used when the processor goes into the subroutine. Channel 1 is the edit line at the bottom of the screen, channel 2 is the main screen and channel 3 is the printer.

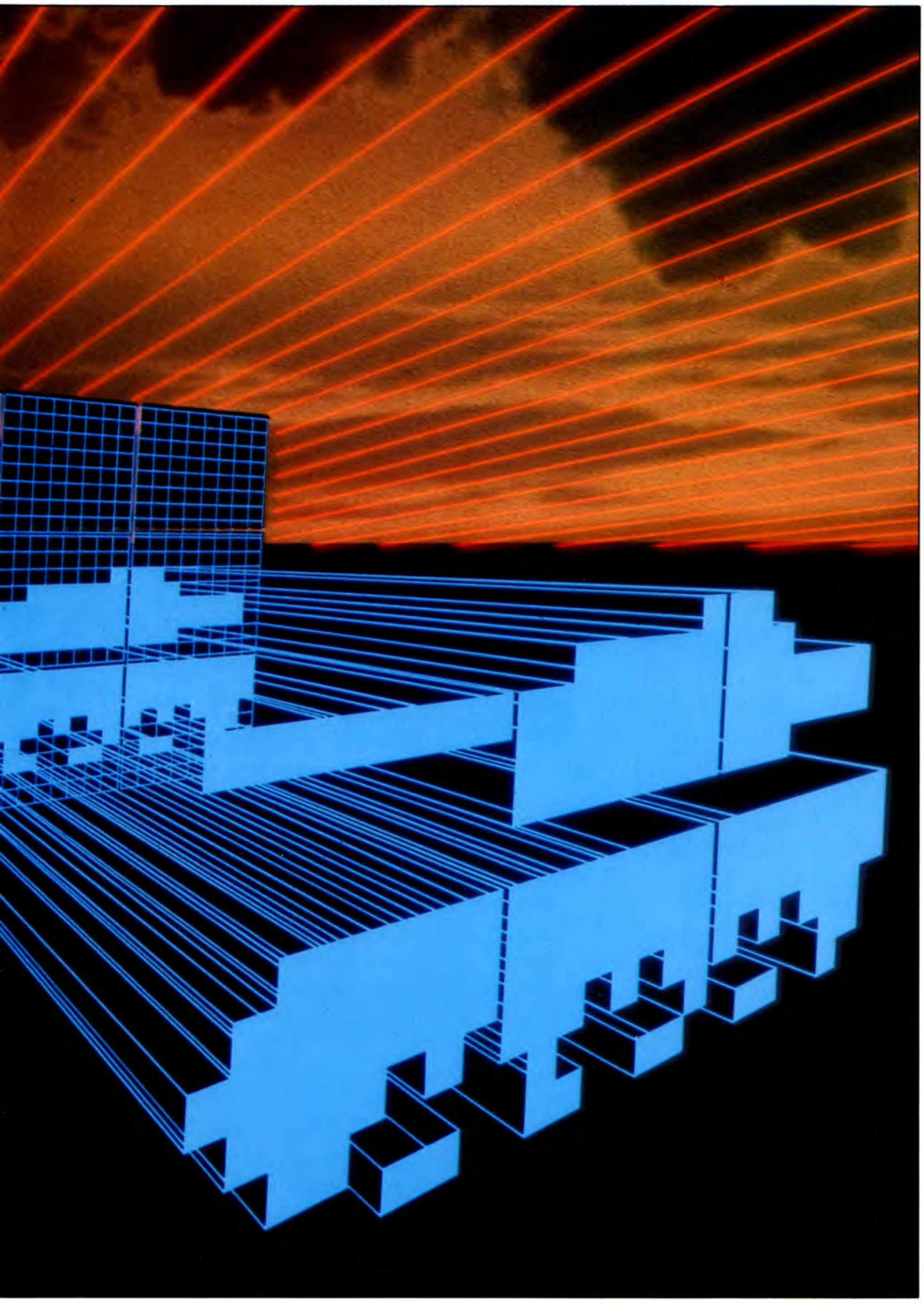
The IX pointer that was **push**ed on the stack is now **pop**ped back into DE. Then BC is loaded with 18 and the string printing routine at 2,Ø32 is **call**ed. This routine prints BC characters, starting at DE. So it prints the 18 characters that make up the frame on the screen starting from the base address that was carried in the IX pointer.

That done, **ret** returns the processor to BASIC.

```
        CLC
        LDA   #&EØ
        DEX
        BMI   ZERO
        BEQ   ONE
        DEX
        BEQ   TWO
        ADC   #&Ø9
TWO     ADC   #&Ø9
ONE     LDX   #&Ø3
AGAIN   LDY   #&Ø3
LINE    JSR   &FFEE
        CLC
        ADC   #&Ø1
        DEY
        BNE   LINE
        DEX
        BEQ   END
        PHA
        LDA   #&ØA
        JSR   &FFEE
        LDA   #&Ø8
        JSR   &FFEE
        JSR   &FFEE
        JSR   &FFEE
        PLA
        JMP   AGAIN
END     RTS
ZERO    LDA   #&2Ø
        LDX   #&Ø3
NEXT    LDY   #&Ø3
```



```
BACK   JSR  &FFEE
       DEY
       BNE  BACK
       DEX
       BEQ  END
       LDA  #&ØA
       JSR  &FFEE
       LDA  #&Ø8
       JSR  &FFEE
       JSR  &FFEE
       JSR  &FFEE
       LDA  #&2Ø
       JMP  NEXT
```



The Acorn routine starts out by clearing the carry flag. There are additions to be done. Then the base address of the UDGs is loaded into the accumulator.

If you look back at the BASIC you will see that the number of the frame required is stored in the resident integer variable X%. This value is transferred into the X register on going into a machine code routine.

DEX decrements the frame number. So if it was Ø, it now goes negative and sets the minus flag. BMI ZERO jumps down to the ZERO routine in that case.

THE ZERO ROUTINE

If the frame number was Ø, the empty frame is required. This is used to overprint the other UDGs to unprint them when they are moved. The first thing the ZERO routine does is load the accumulator with &2Ø, or 32 in decimal. That is the ASCII for a space. The X and Y are loaded with 3—the frame is 3 × 3.

The routine in &FFEE prints whatever is in the accumulator on the screen. And the PRINT TAB in the BASIC program puts the cursor in the appropriate position.

DEY then decrements Y. If the result is not zero, BNE BACK sends the processor back to print another space in the character square—the cursor automatically moves along one space horizontally once it has printed.

So DEY sends the processor back round the BACK loop three times. Then DEX decrements the X register. When this is decremented to zero—and the whole of the frame area has been overprinted by space—BEQ END branches back to the RTS instruction, which returns the processor to BASIC.

When it is not equal to zero though, and there is still some space printing to do, the accumulator is loaded with &ØA, or 1Ø in decimal, which is a line feed. Then the screen print routine at &FFEE is called. When this happens it has the effect of dropping the cursor down on line.

Then the accumulator is loaded with &Ø8, which is a backspace. And the screen print routine is called three times to backspace the cursor to the beginning of the next line of UDGs.

That done, &2Ø, the ASCII for a space, is loaded back into the accumulator and JMP NEXT takes the processor back to load the Y register with 3 again, so that it is ready to print the three characters in the next line of the frame.

PRINTING UDGs

If the frame number is not Ø, the processor does not make the BMI branch. But if it was 1, the DEX would have made it Ø, the zero flag would have been set and the BEQ ONE sends the processor on to the label ONE.

If it's not zero, the X register is decremented again, so if the frame number was 2 to start with, X now contains Ø and BEQ TWO branches the program down to the label TWO. The ADC #&Ø9 then adds 9 onto the base address in A and continues. The frame is 3 × 3, so adding nine moves the pointer carried in A along to the beginning of the next UDG.

If the frame number was 3 to start with, none of the tests above would have picked it up so the processor would move onto the next instruction. This is another ADC #&Ø9, so 9 is added onto the base address twice, which moves the pointer onto the third frame.

X and Y are both loaded with 3, exactly as they were in the unprint routine. Then JSR &FFEE calls the screen print subroutine which prints the first UDG in the top left-hand corner of the frame where the cursor has been put by the BASIC program.

That done, the carry is cleared and 1 is added to the pointer in the accumulator. This moves it along to the next UDG in the frame. Y is decremented to count along horizontally.

BNE LINE sends the processor back to print the next UDG until Y is decremented to Ø and the end of the line has been reached. Then X is decremented. If the result is not Ø, PHA pushes the pointer in the accumulator onto the stack.

The cursor is then moved into the right position to start the next line of UDGs by screen printing a line feed and three backspaces. The pointer is pulled back off the stack and JMP AGAIN takes the processor back to load Y with 3 and starts printing the next line of UDGs.

When X has counted down the three lines, BEQ END branches to RTS, which returns the processor to BASIC.

```
        ORG  32ØØØ
        LDX  327ØØ
        LDA  #3
        STA  FCNT
        STA  SCNT
        LDA  #8
        STA  TCNT
        LDA  3225Ø
        BEQ  JUMP
        LDU  #323ØØ
        DECA
        LDB  #72
        MUL
        LEAU D,U
LOOP    LDA  ,U+
        STA  ,X
        LEAX 32,X
        DEC  TCNT
        BNE  LOOP
        LDA  #8
        STA  TCNT
        LEAX -255,X
        DEC  FCNT
        BNE  LOOP
        LDA  #3
        STA  FCNT
        LEAX 253,X
        DEC  SCNT
        BNE  LOOP
        RTS
JUMP    CLRB
STLP    STB  ,X
        LEAX 32,X
        DEC  TCNT
        BNE  STLP
        LDA  #8
        STA  TCNT
        LEAX -255,X
        DEC  FCNT
        BNE  STLP
        LDA  #3
        STA  FCNT
        LEAX 253,X
        DEC  SCNT
        BNE  STLP
        RTS
FCNT    RMB  1
SCNT    RMB  1
TCNT    RMB  1
```

The screen address of the top left-hand corner of the grid is POKEd into memory locations 32,7ØØ and 32,7Ø1 by the BASIC program. And LDX loads this two-byte pointer into the 16-bit X register. The number 3 is loaded into the accumulator and stored in the two counters labelled FCNT and SCNT. The frame contains 3 × 3 UDGs.

These counters appear in the data table at the end of program. Here the space the counter is going to occupy is reserved by the instruction RMB—Reserve Memory Byte. The number 1, following, tells it to reserve just one memory byte. RMB 5Ø would reserve fifty memory bytes.

The third counter, TCNT, is set to 8. Then the contents of 32,25Ø are loaded into A. The number of the UDG is POKEd into 32,25Ø. If the UDG's number is zero, BEQ JUMP sends it off to the routine which clears that area of the screen. Otherwise the processor goes on to deal with the UDG itself.

PRINTING THE GRAPHIC

Memory location 32,3ØØ is the start of the UDG store. And the number 32,3ØØ is loaded into U so that you can work out where the appropriate UDG begins. The required UDG's number in the accumulator is decremented—the first UDG starts at the beginning of store, so its offset is Ø.

LDB #72 loads the B register with the number 72, and MUL MULtiplies the contents of A and B then puts the result in the D register. The accumulators A and B are eight-bit registers and D is a 16-bit register, so it won't overflow. There are 72 bytes in each graphic. Each UDG contains eight bytes and there are nine UDGs in the frame—9 × 8 = 72. So this operation works out the offset required to find the beginning of the appropriate UDG.

LEAU D,U loads U with the value of D plus U—in other words, it counts along the table until the beginning of the required UDG is pointed to.

LDA ,U+ loads the accumulator with the contents of that memory location, then increments the U register ready to deal with the next one. And STA ,X stores the byte of the UDG picked up from the graphics table into the memory location pointed to by X. The X register, you'll recall, carries the screen location of the top left-hand corner of the graphic.

LEAX 32,X then adds 32 to the X register. This moves the X pointer down the screen one line—there are 32 character squares on each line, remember. The TCNT counter is then decremented. And if it is not zero the BNE instruction following it loops back to deal with the next byte of the UDG.

The initial value of TCNT was 8, so the loop is executed eight times. It takes eight bytes, one above the other, to form a UDG. So this counter counts out the bytes that form the entire UDG.

ADDING THE NEXT UDG

LDA #8 and STA TCNT sets the TCNT counter back to eight, ready to deal with the next UDG. Then LEAX -255,X moves the X pointer onto the beginning of the next block to the right.

The X register contains the screen pointer, remember. It has been incremented by 32 eight times to print out the first UDG—32 × 8 = 256. But you want to move back to the position one to the right of the screen position of the start of the first UDG, so you only have to wind the X pointer back 255.

The FCNT counter is decremented so that it counts across the array of three UDGs side-by-side. When the result of the decrementing is not zero, the processor loops back to start dealing with the first byte of the next UDG. When the result is zero, the processor moves on.

MOVING DOWN THE FRAME

The FCNT counter is then reset to 3 and the X pointer is wound forward by 253. Remember that the X pointer has already been wound forward by 32, which took it to the screen location directly beneath the UDG just printed. Then it was wound back by 255 to take it to the screen location immediately to the right of the top of the last UDG.

If you draw a little grid showing the UDGs and look at the relative positions of the screen locations, you will see that after the X register has been wound forward by 32—and points to the screen location directly beneath the UDG just printed—it is two spaces to the right of where the first byte of the first UDG in the second row should be. So at that point, 2 should be subtracted from the X pointer to move it to the right position.

But 255 has already been subtracted by the program, so if you add 253 you get back to the right place.

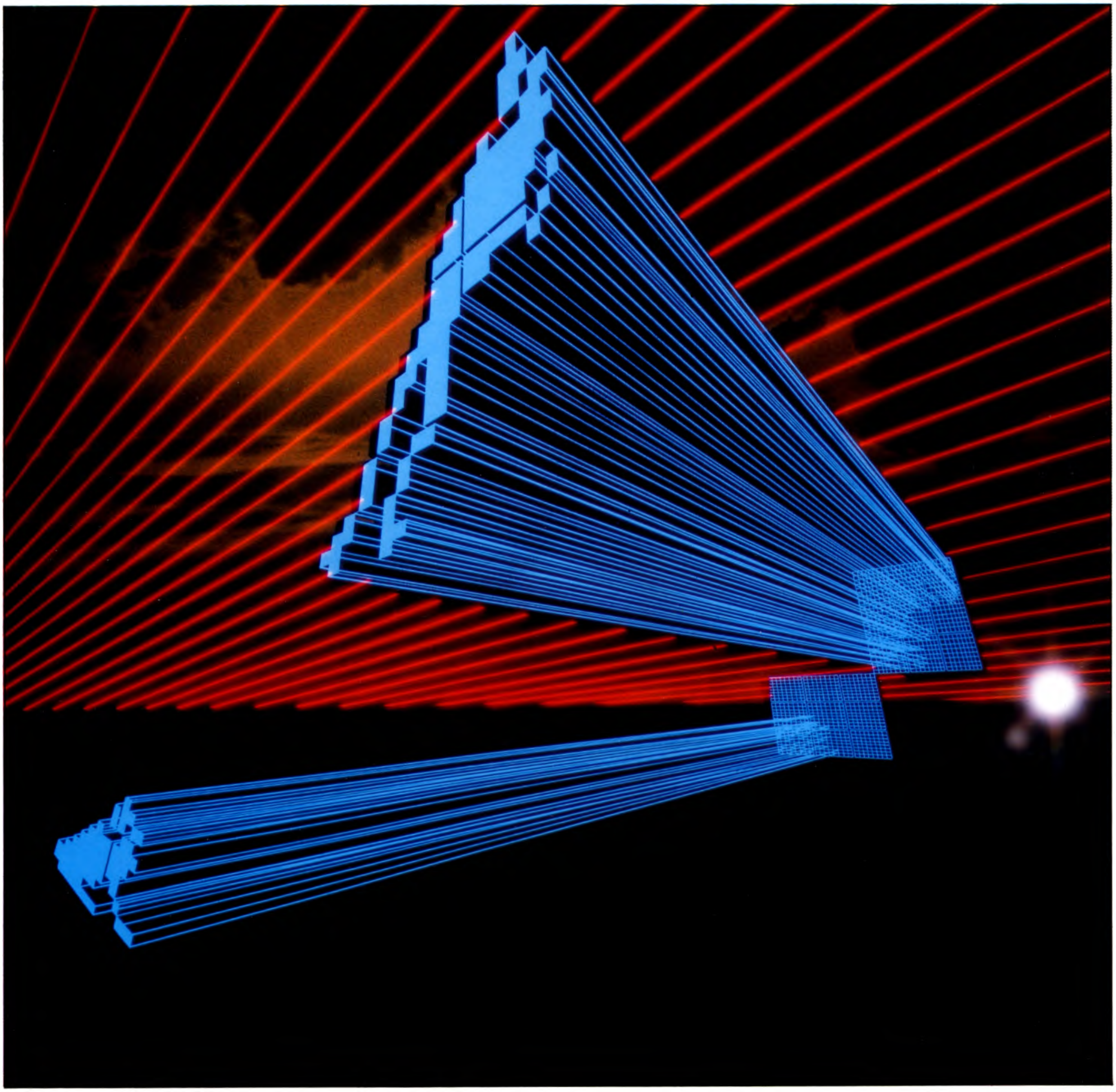
The SCNT counter is decremented. This one counts down the three lines of UDGs which make up the whole graphic. And when the result of the decrementing is not zero, the BNE takes the processor back to deal with the next line.

If the result is zero, though, the whole of the graphic has been printed on the screen and RTS takes the processor back to BASIC.

UNPRINTING ON THE SCREEN

When the number of the graphic to be printed on the screen is zero, the routine clears the area of the screen being addressed. Clearing the screen is done in much the same way as printing on it. Only the routine does not have to look up the UDG table. It just has to print the bit pattern for Ø—in other words, nothing—on the screen.

CLRB sets the B register to Ø. The Ø is then stored in the screen position pointed to by the X register. The X register is then updated repeatedly in exactly the same way as the print routine to move it location by location across the whole grid. Only this time the contents of the B register—Ø—are stored in each location. And again, when it has finished the RTS returns it to BASIC again, ready to run the rest of your program.



HOW COMPUTERS STORE NUMBERS

Although you program in BASIC, the computer works in numbers. If you know how numbers are stored, you can save memory, and even get more accuracy . . .

If your computer performs a numeric calculation that has a very large or very small result, you may see strange looking numbers on your computer's TV screen. An example of the sort of answer that may be PRINTed is 2.34E14.

What this means is that the number consists of two parts—one giving the digits in the number (the 2.34 in the example above) and the other (the E14 in the example above) telling the computer (and you) where the decimal point should be.

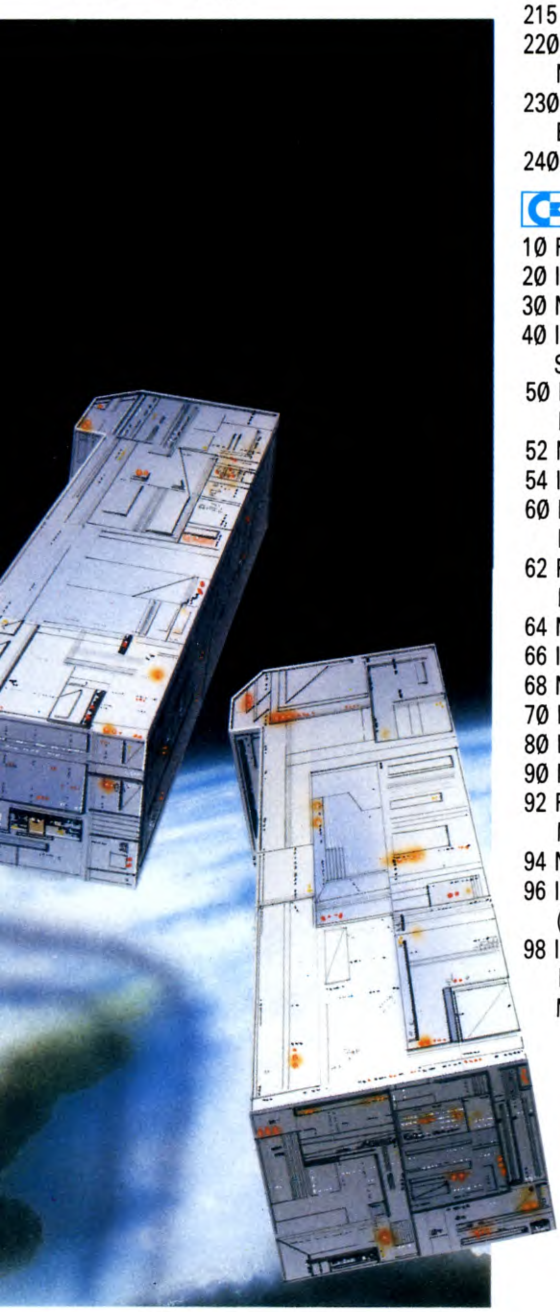
The reason that you see numbers in this form from time to time is because of the way that numbers are stored in your computer; there is a limit to the size of the numbers that it PRINTs. The Spectrum, for example, can only PRINT numbers of up to 8 digits: if your number has more digits than this, the computer splits it up into something which shows the significant digits, plus something which shows how big it is, and PRINTs it in the form shown above. 2.34E14 actually means 234ØØØØØØØØØØØØ—too large to PRINT out.

You can also use the 'E' form yourself as a shorthand when you enter numbers in a program. This program will help you to understand how this form of numbers (often called 'exponent') works, and how the E value is calculated.

```
10 CLS: POKE 23658,8
2Ø INPUT "INPUT NUMBER", LINE A$
25 IF A$="" THEN GOTO 2Ø
3Ø LET N$="Ø": LET E=Ø: LET N=VAL A$
4Ø IF N=Ø THEN PRINT "VALUE TOO
   SMALL": PAUSE 5Ø: GOTO 1Ø
5Ø LET F=Ø: FOR M=1 TO LEN A$: IF
   A$(M)="E" THEN LET F=M
55 NEXT M: IF F=Ø THEN GOTO 2ØØ
6Ø LET N$=STR$ N: LET F=Ø: FOR M=1
   TO LEN N$: IF N$(M)="E" THEN LET
   F=M
65 NEXT M: IF F=Ø THEN GOTO 18Ø
68 LET N$=N$ (TO F-1)
7Ø IF ABS N<1 THEN GOTO 13Ø
8Ø IF ABS N<1Ø THEN GOTO 11Ø
9Ø LET N=N/1Ø: LET F=Ø: FOR M=1 TO
   LEN N$: IF N$(M)="." THEN LET F=M
92 NEXT M: IF F=LEN N$ THEN LET
   N$=N$ (TO LEN N$-1): GOTO 1ØØ
95 IF F< >Ø THEN LET N$=N$ (TO
   F-1)+N$(F+1)+"."+N$(F+2 TO ):
   GOTO 8Ø
1ØØ LET N$=N$+"Ø": GOTO 8Ø
11Ø IF N$(LEN N$)="." THEN LET N$=N$
   (TO LEN N$-1)
12Ø GOTO 18Ø
13Ø IF ABS N> =1 THEN GOTO 17Ø
14Ø LET N=N*1Ø: LET F=Ø: FOR M=1 TO
   LEN N$: IF N$(M)="." THEN LET F=M
145 NEXT M: IF F=1 THEN LET
   N$=".Ø"+N$(2 TO ): GOTO 13Ø
```

- ■ EXPONENTS
- ■ HOW COMPUTERS STORE NUMBERS
- ■ A FLOATING POINT
- ■ NEGATIVE NUMBERS
- ■ PEEKING YOUR COMPUTER'S MEMORY
- ■ TIPS TO SAVE MEMORY
- ■ NUMBER FORMATTING ON ACORN DRAGON AND TANDY COMPUTERS

```
150 IF F< >Ø THEN LET N$=N$( TO
    F−2)+"."+N$(F−1)+N$(F+1 TO ):
    GOTO 13Ø
16Ø LET N$="."+N$: GOTO 13Ø
17Ø IF VAL A$<Ø THEN LET
    N$="−"+N$
```



```
18Ø PRINT : PRINT A$;"□EQUALS": PRINT
    N$
19Ø GOTO 2Ø
2ØØ IF ABS N<1 THEN GOTO 22Ø
21Ø IF ABS N>=1Ø THEN LET E=E+1:
    LET N=N/1Ø: GOTO 21Ø
215 GOTO 23Ø
22Ø IF ABS N<=1 THEN LET E=E−1: LET
    N=N*1Ø: GOTO 22Ø
23Ø PRINT A$;"□EQUALS": PRINT N;: IF
    E< >Ø THEN PRINT "E";E
24Ø PRINT : GOTO 2Ø
```

```
1Ø PRINT "♥"
2Ø INPUT "▚INPUT NUMBER"; A$:PRINT
3Ø N$="Ø":E=Ø:N=VAL(A$)
4Ø IF N=Ø THEN PRINT "VALUE TOO
   SMALL▚":GOTO 2Ø
5Ø FOR F=1 TO LEN(A$):IF
   MID$(A$,F,1)="E" THEN 54
52 NEXT F:F=Ø
54 IF F=Ø THEN 2ØØ
6Ø N$=RIGHT$(STR$(N),
   LEN(STR$(N))−1)
62 FOR F=1 TO LEN(N$):IF
   MID$(N$,F,1)="E" THEN 66
64 NEXT F:F=Ø
66 IF F=Ø THEN 18Ø
68 N$=MID$(N$,2,F−2)
7Ø IF ABS(N)<1 THEN 13Ø
8Ø IF ABS(N)<1Ø THEN 11Ø
9Ø N=N/1Ø
92 FOR F=1 TO LEN(N$):IF
   MID$(A$,F,1)="." THEN 96
94 NEXT F:F=Ø
96 IF F=LEN(N$) THEN N$=LEFT$
   (N$,LEN(N$)−1):GOTO 1ØØ
98 IF F< >Ø THEN N$=LEFT$(N$,
   F−1)+MID$(N$,F+1)+"."+
   MID$(N$,F+2):GOTO 8Ø
    1ØØ N$=N$+"Ø":GOTO 8Ø
    11Ø IF RIGHT$(N$,1)="." THEN
        N$=LEFT$(N$,LEN(N$)−1)
    12Ø GOTO 18Ø
    13Ø IF ABS(N)>=1 THEN 17Ø
    14Ø N=N*1Ø
    142 FOR F=1 TO LEN(N$):IF
        MID$(N$,F,1)="." THEN 146
    144 NEXT F:F=Ø
    146 IF F=1 THEN N$=".Ø"+MID$
        (N$,2):GOTO 13Ø
15Ø IF F< >Ø THEN N$=LEFT$(N$,
    F−2)+"."+MID$(N$,F−1,1)+
    MID$(N$,F+1):GOTO 13Ø
16Ø N$="."+N$:GOTO 13Ø
17Ø IF VAL(A$)<Ø THEN N$="−"+N$
18Ø PRINT "♥";A$;"▙EQUALS":
    PRINT N$
19Ø GOTO 2Ø
2ØØ IF ABS(N)<1 THEN 22Ø
21Ø IF ABS(N)>=1Ø THEN E=E+1:
    N=N/1Ø:GOTO 21Ø
215 GOTO 23Ø
22Ø IF ABS(N)<=1 THEN E=E−1:
    N=N*1Ø:GOTO 22Ø
23Ø PRINT "♥▙";A$;"▙EQUALS":
    PRINT N;"▌▌";:IF E< >Ø THEN PRINT
    "E";E
24Ø PRINT:GOTO 2Ø
```

```
1Ø INPUT A$
2Ø P=INSTR(A$,"E")
3Ø A$=STR$(EVAL(A$))
4Ø @%=(LEN(A$)+(INSTR(A$,
   ".")< >Ø)−(LEN(A$)=1))
   *256+&1ØØØA
5Ø IF P□AND INSTR(A$,"E")=Ø THEN
   @%=1Ø:GOTO 7Ø
6Ø IF P□THEN PROCNORM:GOTO 1Ø
7Ø PRINT;EVAL(A$)
8Ø GOTO 1Ø
9Ø DEF PROCNORM
1ØØ IF INSTR(A$,".") THEN
    A$=LEFT$(A$,1)+RIGHT$
    (A$,LEN(A$)−2)
11Ø P=INSTR(A$,"E")
12Ø E=EVAL(RIGHT$(A$,LEN (A$)−P))
13Ø A$=LEFT$(A$,P−1)
14Ø IF E<Ø THEN 18Ø
15Ø E=E−LEN(A$)+1
16Ø A$=A$+STRING$(E,"Ø")
17Ø GOTO 19Ø
18Ø A$="Ø."+STRING$(−E−1,
    "Ø")+A$
19Ø PRINTA$
2ØØ ENDPROC
```

```
1Ø CLS
2Ø PRINT:INPUT" INPUT NUMBER□";A$
```

```
3Ø N$=“Ø”:E=Ø:N=VAL(A$)
4Ø IF N=Ø THEN PRINT“ VALUE TOO
   SMALL ”:PRINT:GOTO2Ø
5Ø F=INSTR(A$,“E”):IF F=Ø THEN2ØØ
6Ø N$=STR$(N):F=INSTR(N$,“E”):
   IFF=Ø THEN 18Ø ELSEN$=
   MID$(N$,2,F-2)
7Ø IF ABS(N)<1 THEN 13Ø
8Ø IF ABS(N)<1Ø THEN 11Ø
9Ø N=N/1Ø:F=INSTR(N$,“.”):
   IF F=LEN(N$) THEN N$=LEFT$
   (N$,LEN(N$)-1) ELSE IFF<>Ø
   THENN$=LEFT$(N$,F-1)+
   MID$(N$,F+1,1)+“.”+
   MID$(N$,F+2):GOTO8Ø
1ØØ N$=N$+“Ø”:GOTO8Ø
11Ø IFRIGHT$(N$,1)=“.” THEN
   N$=LEFT$(N$,LEN(N$)-1)
12Ø GOTO18Ø
13Ø IF ABS(N)>=1 THEN 17Ø
14Ø N=N*1Ø:F=INSTR(N$,“.”):
   IFF=1 THENN$=“.Ø”+MID$
   (N$,2):GOTO13Ø
15Ø IF F<>Ø THENN$=LEFT$(N$,
   F-2)+“.”+MID$(N$,F-1,1)+
   MID$(N$,F+1):GOTO13Ø
16Ø N$=“.”+N$:GOTO13Ø
17Ø IF VAL(A$)<Ø THENN$=“-”+N$
18Ø PRINT:PRINTA$;“□EQUALS”:
   PRINTN$
19Ø GOTO2Ø
2ØØ IF ABS(N)<1 THEN 22Ø
21Ø IF ABS(N)>=1Ø THEN E=E+1:
   N=N/1Ø:GOTO21Ø ELSE 23Ø
22Ø IF ABS(N)<=1 THEN E=E-1:
   N=N*1Ø:GOTO22Ø
23Ø PRINTA$;“□EQUALS”:PRINTN;
   CHR$(8);:IF E<>Ø THEN
   PRINT“E”;E
24Ø GOTO 2Ø
```

When you RUN these programs, the computer waits for you to ENTER a number. You should type in a positive number (or negative on Dragon and Tandy) in either normal form, or an exponent number—don't worry if you still do not know what an exponent number is, you soon will. The computer then PRINTs your number in the opposite form. The Acorn and Dragon have commands you can use to change the format in which the computers PRINT the numbers. These are explained later in this article.

To convert a number written as an exponent to one which is more meaningful to you, you need simply to multiply the digits part of the number (known as the 'mantissa') by 10 to the power of the number after the letter E. The number after the letter E, is known as the 'exponent' of the number.

Although this sounds quite complicated, if you work through it step by step, it soon becomes clear. Take the number 1.23E4.

To convert it to the usual form, multiply the mantissa (1.23), by 10 to the power of the exponent—which gives the sum 1.23*10,000. So the exponent number 1.23E4 becomes 12,300 in normal form.

You can experiment with converting numbers like this, using the programs above to check your answers. Notice that for any number, the mantissa always takes a value with one digit before the decimal point, so it varies between 1.0 and 9.99999 . . .

Negative numbers are very similar: instead of having a positive number as the mantissa of the exponent number, you have a negative value. (You should note that putting a minus sign after the E part has a very different effect; try this with the programs above to see what happens.)

NUMBER STORAGE

While knowing about exponent numbers like this may be interesting, it is also useful as it shows the way in which computers (at least, all the computers covered here and most others, too) store numbers.

When you type in a simple direct command like PRINT 1Ø*1Ø, the computer actually deals with it in rather more detail: it translates the numbers into a type of exponent (called 'floating point') format, and calculates them in this form.

There is also another complication in the way that most computers store numbers. As you have already seen from creating UDGs in BASIC programming, and from machine code programming, the computers store each number in binary, or base two.

To show how numbers are stored in your computer's RAM, follow this example of a calculation based on the number 10.

The first step is to convert it into binary—which gives the number ØØØØ1Ø1Ø.ØØØØØ . . .

The first four zeros of this binary number mean nothing at all, and could quite easily be missed out: which leaves the binary number 1Ø1Ø.ØØØØØ . . .

As you saw earlier in this article, a decimal exponent number consists of a number between 1.0 and 9.999999. . . when a floating point number is stored in your computer's memory, the mantissa consists of a binary number which always starts .1.

This is possible because the size of the exponent part of the number determines the position of a 'binary point' (the binary equivalent of a decimal point). As it is automatically defined, the mantissa does not need to specify where the point is.

So there is a problem: how do you set the exponent to show where the binary point should be, and how do you convert the binary number so that it starts .1?

MOVING THE POINT

In fact, the answer to both problems is the same. All you do is move the point until it is just in front of the first 1, and for every place that you move the point, you add one to the exponent. For example, with the number 58 (decimal) or 111Ø1Ø.ØØØ (binary), the binary point is gradually moved to the left, until there are no numbers to the left of it, or only zeros. In this case you have to move the binary point 6 places to the left—so the exponent is +6 and the mantissa is .111Ø1ØØØ . . .

In fact, the exponent starts off as 128, for reasons which you will discover later in this article, and so you actually add the number to this. Therefore, the exponent in the example above, for the number 58 is 128+6, or 134 (and, of course, the computer stores this as a binary number).

To recap, the computer stores the exponent part of your number in one byte, and it also stores the mantissa part of your number. In fact, the mantissa part always takes up four

bytes, and so in the example above, three of these bytes would be filled with zeros.

Any floating point number thus takes up a total of five bytes. This limits the size of the maximum number that the computer can store. The single byte of the exponent gives it a maximum possible value of 2 to the power of 127—it would be to the power of 256, except that the first bit is used as a sign bit (to tell whether the exponent is positive or negative) leaving only seven bits for the exponent.

And the maximum possible number for the mantissa is .11111 etc, where every bit is set to 1. This is very close to 1.ØØØØ etc, which is 1 in either binary *or* decimal. The smallest possible number is .1ØØØ etc, when all but the first bit is Ø. (The first bit has to be one because the binary point is moved until it reaches the first one in the number.) And .1 in binary is $\frac{1}{2}$ in decimal.

By multiplying the two parts of the number, you can find the maximum possible number that the computer can store in floating point form—this is 1.7Ø141E38. You can use the programs later on in this article to check how your computer stores this number.

Unlike the other computers, the Spectrum actually needs *six* bytes to store floating point numbers. Five of these are identical to those on other machines. The sixth is necessary because the computer needs to be told to expect a floating point number. This is done by preceding each one by a number—decimal 14—which also takes up one byte.

SPECIAL CASES

You now have a good idea of how the computer stores a number in floating point form. But there are two variations on this.

The first comes when your initial number is less than one. When this is the case, moving the point to the left actually makes it further away from where you want it to be! So you simply move it to the right instead. Also, so that the computer knows where the point should be in the final number, subtract one from the exponent every time you move the point, instead of adding one. The exponent starts, as with numbers greater than one, at 128.

Often, when you write a binary number, the decimal equivalents of each 'column' of the binary number are written at the top. You have seen this before (for example in UDGs) for the numbers to the left of the point, but not for those to the right of the point. Those to the right are actually fractions, which might at first seem a bit strange, but this is perfectly logical.

In binary, to find the decimal value of the next column on the left, you multiply by two—so 1 becomes 2, 2 becomes 4, and 4 8, and so on. Working from left to right, therefore, you do the opposite—divide by 2. And 1 divided by 2 is a half, half divided by two is a quarter, and so on.

The second variation with floating point numbers is when the whole number is less than 0, in other words, it becomes negative. When this happens, the computer must be able to store its own version of a minus sign somehow—and it manages to do this without using up any more memory space.

You saw earlier that, because of the way the numbers are stored, the first bit of the first byte of the mantissa of your number must always be 1. Knowing this, the computer assumes it to be the case and actually uses this bit to tell whether the number is positive or negative. Simply, if the number is negative, this bit is set to 1; and if it is positive then the bit is set to Ø.

WHERE'S THAT NUMBER?

With the examples you saw earlier in mind, you are now in a position to be able to know how the computer stores your numbers.

The decimal number 58 (111Ø1Ø in binary) is stored as the following 5 bytes:

134 1Ø4 Ø Ø Ø

and the number 10 is held in these bytes

132 32 Ø Ø Ø

Try some experiments, working out the decimal values of the bytes that the computer stores for your numbers. You can test your answers by typing in and RUNning this program. Just type RUN 1ØØ to use the second (or GOTO 1ØØ on the Acorns).

While the Spectrum stores numbers in the general way described above, it is just one of two ways that the computer actually uses to store numbers. If you use a whole number between −65535 and +65535, the Spectrum stores it in a different form—simply, the binary equivalent—and in just two bytes. Although this would appear to save memory, unfortunately, the Spectrum also uses three empty bytes, so that the whole number takes up the same amount of memory as the floating point number.

For this reason, when you use the first of the two programs below, you should not enter integers between −65535 and +65535. If you do, the program will still work, but the results will not correspond with the explanation above.

```
1Ø BORDER 1: PAPER 7: INK 9: CLS
2Ø INPUT "Enter a number (not an integer)",x
```

```
40 PRINT '"Exponent:□";PEEK (PEEK
   23627+256*PEEK 23628+1)
50 PRINT '"Mantissa:";: FOR n=2 TO 5
60 PRINT TAB 10;PEEK (PEEK
   23627+256*PEEK 23628+n)
70 NEXT n
80 STOP
```

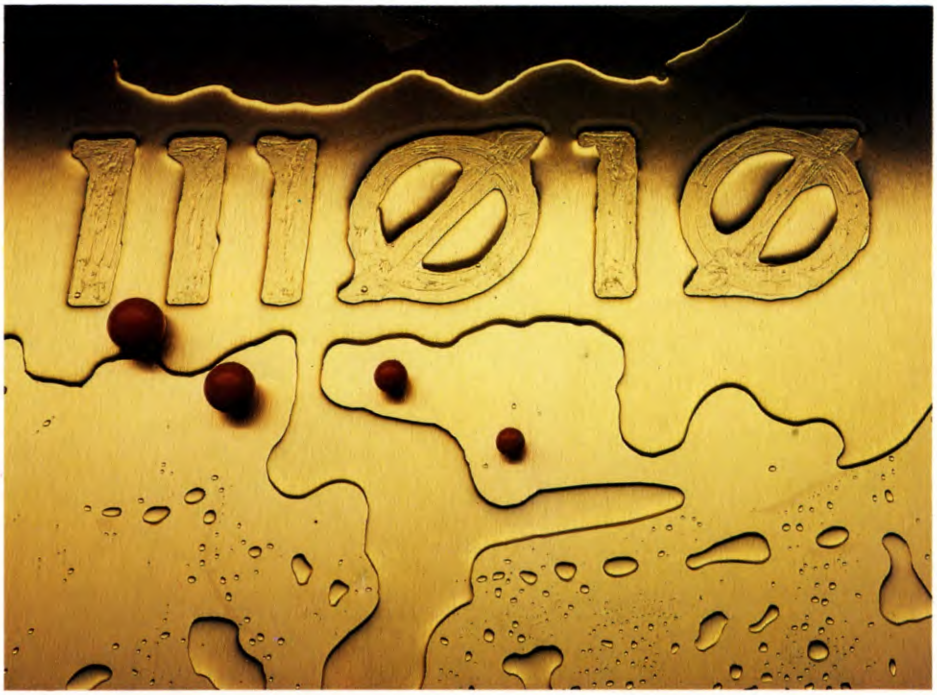
```
10 PRINT "♥":CLR
20 PRINT "ENTER A NUMBER":
   INPUT X
30 V=PEEK(45)+PEEK(46)*256
40 PRINT "EXPONENT:";PEEK(V+2)
50 PRINT "MANTISSA:";:FOR N=3 TO 6
60 PRINT TAB(10);PEEK(V+N)
70 NEXT N
80 END
```

```
10 @%=10
20 REPEAT
30 INPUT'"ENTER A NUMBER",A
40 PRINT'"THESE NUMBERS MAKE
   UP□";A''
50 PRINT"EXPONENT□□□□□□□□";
   ?(LOMEM+3)'"MANTISSA";
60 FOR T=LOMEM+4 TO LOMEM+7:
   PRINTTAB(15);?T:NEXT
70 UNTIL Ø
```

```
10 CLS
20 INPUT"□INPUT□A□NUMBER□";N
30 D=VARPTR(N)
40 PRINT:PRINT"□EXPONENT
   □=□",PEEK(D)
50 PRINT"□MANTISSA□=□";
60 FORG=1TO4
70 PRINT,PEEK(D+G)
80 NEXT
90 PRINT:GOTO20
```

```
100 REM SECOND PROGRAM
110 BORDER 1: PAPER 7: INK 9: CLS : LET
    r=Ø
120 INPUT AT 1,Ø;"Enter exponent",exp
130 IF exp<Ø OR exp>255 THEN GOTO 120
140 PRINT '"Exponent:□";exp: POKE PEEK
    23627+256*PEEK 23628+1,exp
150 PRINT '"Mantissa:";: FOR n=2 TO 5
160 INPUT AT 1,Ø;"Enter mantissa", man
170 IF man<Ø OR man>255 THEN GOTO160
180 PRINT TAB 10;man: POKE PEEK
    23627+256*PEEK 23628+n,man
190 NEXT n
200 PRINT '"Result=□";r
```

```
100 REM SECOND PROGRAM
110 PRINT "♥":CLR:R=1:
    V=PEEK(45)+PEEK(46)*256
120 INPUT "ENTER EXPONENT";EX
130 IF EX<Ø OR EX>255 THEN 12Ø
140 POKE V+2,EX
150 FOR N=3 TO 6
160 INPUT "ENTER MANTISSA";MAN
170 IF MAN<Ø OR MAN>255 THEN 16Ø
180 POKE V+N,MAN
190 NEXT N
200 PRINT "RESULT=";R
```



```
100 REM SECOND PROGRAM
110 A=Ø
120 REPEAT
130 INPUT'"EXPONENT□",?(LOMEM+3)'
140 FOR T=4 TO 7
150 PRINT"MANTISSA BYTE□";T-3;:
    INPUT?(T+LOMEM)
160 NEXT
170 PRINT'"THESE BYTES MAKE□";A
180 UNTILØ
```

```
100 REM SECOND PROGRAM
110 CLS
120 N=1:D=VARPTR(N)
130 INPUT"□INPUT EXPONENT□";E
140 POKED,(255ANDE)
150 PRINT"□INPUT MANTISSA□";
160 FORK=1TO4
170 INPUT E
180 POKED+K,(255ANDE)
190 PRINT,;
200 NEXT
210 PRINT:PRINT"□NUMBER IS□"; N:PRINT
220 GOTO 130
```

The first of these two programs lets you INPUT a number, and then the computer PRINTs the decimal values of the five bytes that are used to store the number that you enter.

The second lets you enter five bytes, and the computer then PRINTs out the decimal number that they represent (some of the answers to this may be in exponent form).

SAVING MEMORY

Knowing how the computers store numbers, in five bytes, you will realise that it is often very wasteful on memory since for numbers which do not need all four bytes of the mantissa, the bytes could be saved and perhaps used for something else.

In fact, you can save a significant amount of memory by avoiding the use of *numbers* altogether, but, of course, this is not always possible. There is also, however, a way to reduce the amount of memory that each number takes up.

You can avoid using numbers in many cases by setting up a variable as being equal to the value you want to use. Of course, you need to use the number itself when you define the variable, and the variable itself will also take several bytes to store—so if you are only going to use the number once or twice then it is probably not worthwhile.

Once you have set up a variable, though, whenever you use it instead of a number, you need just one byte for every character of the variable. Remember that if you use a variable with a long name, you are unlikely to save much memory, but if you use just one letter for the variable name, you are likely to save quite a lot.

Some numbers, especially 1 and 0, both crop up so often in almost all programs that it is generally good programming practice to use a variable for them.

This method is especially useful on the Spectrum computer, because it uses six bytes for every floating point number instead of the five that the other computers take.

ODD EFFECTS

When you understand how the computer stores numbers, you can also find out why some strange occurrences seem to happen. Try these short examples for your machine. The reasons behind them are explained later.

First, try this Line:

```
PRINT 1Ø.ØØØØØØØØØ1
```

You can see that the computer chops off the last digit, the 1, and converts your number to just 1Ø.

This may be a problem when you want high accuracy, but you can use this to your advantage with the program above which PRINTed out the 5 bytes in which your number is stored. This did not permit you to see the flashing point version of values between −65535 and +65535 because the Spectrum stores these in a different way. But, if you enter a string of Ø's after a decimal point, followed by a 1, the computer stores your number in full floating point form.

Some programs use VAL and a number in quotes instead of a normal number. Why is this?
This form is sometimes used to try to save memory space. This is because the computer uses about seven bytes of memory to store most numbers, which can be wasteful. If you put long numbers in a string, and use the function VAL to evaluate it, this effectively makes the string a number, and uses less memory—just one byte for each digit, one for each quote sign, and one for the VAL function.

So, suppose you wanted to find what 5 bytes make up the decimal number 10 in the computer's memory. If you enter 1Ø to the program, the results will not show you the floating point bytes which make up 10. But you can get these if you enter 1Ø.ØØØØØØØØØ1, which the computer does store as a floating point number. And because it chops off the last digit, this number is also the same as 10 to the computer—so you could then get the bytes which combine to form 10.

Now try entering this line:

```
PRINT INT −65536
```

This in fact exposes a weakness of the Spectrum's ROM, which does not differentiate between this value and −1.

Try this direct command on your computer:

```
T=Ø: FOR P=Ø TO 1ØØØØØ: T=T+
Ø.ØØØØØØ1: PRINT T: NEXT
```

If you now press RETURN, and watch the left hand columns of numbers rush up the screen, you should be able to spot the error. What in fact happens is that the seventh digit after the decimal point grows by 1 every time that the loop is executed—and, every now and then, so does the eighth digit—for no apparent reason.

You can see the strange effects on the Dragon and Tandy by entering these two numbers into the program above (the first of the article):

```
123456789Ø1 E−4
```

and:

```
454545454 E−46
```

The first number's discrepancy is quite obvious, and the second number's last digit is actually changed by the computer.

These inaccuracies are not, as some people think, bugs in the computer's ROM, except for the second Spectrum example. The reason why the 1's appear or disappear, seemingly at random, is that the computers will work to a limited degree of accuracy—and so for numbers with more than a certain number of digits, the computers have to round up.

When this happens, depending upon what the numbers are, and upon how many calculations the computer subsequently does with the rounded up values, the results can be wrong—as the program examples show.

Of course, most of the time the computers are quite accurate enough, and these 'errors' do not happen. But, for more serious applications, such as accounts programs, the inaccuracies might be a problem, so the better software packages take this into account.

By using arrays, you can store numbers with more significant figures than the maximum of nine that each of the computers covered here allows. The main advantage with this is that you can then write in a routine to PRINT the number in its normal form, and not as an exponent, which the computer would do. In this way you can also get round the problem of error messages such as 'number too large'.

NUMBER FORMATTING

Earlier, it was explained that the Acorn, Dragon and Tandy have commands which you can use to change the PRINTing format for numbers.

The Acorn computers let you change the format of PRINTed numbers. One of the aspects you can change is 'field width'. To see the effect of the field width, enter the following lines:

```
PRINT 1,2,3,4,5,6,7,8,9
PRINT 123456789
```

The first line PRINTs each figure separately, whereas the second line PRINTs all the figures as one number. By separating each figure or number with a comma, you can PRINT columns of data, but you need not stick to the width you're given when the computer is switched on. There is a special variable called @% which can alter the display. It can be set to change the width of the print fields to any number of columns you like, and it can also alter how numbers are PRINTed out.

One useful setting is @%=&2Ø2ØA (don't worry, the numbers are explained below) which makes sure that all numbers PRINTed out have two figures after a decimal point. This is ideal for amounts of money or any metric weights and measures.

WHAT THE NUMBERS MEAN

When the computer is first switched on, @% equals &9ØA or, to write it out in full, @%=&ØØØØØ9ØA. Note the & sign; this indicates that the number is in hexadecimal. It is best to think of the number as divided into three pairs or bytes, so @%=& byte 4 byte 3 byte 2 byte 1. (Byte 4 is, in this case, unused.)

Byte 3 can be set to ØØ, Ø1 or Ø2 and it determines how numbers are printed out, that is, their format. (You can miss out the first Ø.) Byte 3 equals Ø in the normal format—numbers up to nine digits are printed out normally and numbers with more than nine digits are printed in scientific notation. Try it for yourself. Type in PRINT 123456789 and it will print it out as you wrote it. But type in PRINT 123456789123 and it will print out 1.23456789E11.

If Byte 3 is equal to 1 (format 1) then *all* numbers are PRINTed in scientific notation. Byte 3 equal to 2 (format 2) is the most useful for our needs. This is the one that PRINTs all numbers with a fixed number of digits after the decimal point.

Byte 2 can be any number, but its effect depends on what type of format you chose with Byte 3. With the normal format—format Ø—Byte 2 gives the maximum number of digits printed before reverting to scientific notation. This is set to Ø9 when the computer is switched on, so you get a maximum of nine digits.

With format 1, Byte 2 gives the number of digits shown before the E part, which determines the accuracy of the number.

With format 2, Byte 2 gives the number of digits after the decimal point. Byte 2 is often set to Ø2 to give two decimal figures.

Finally, Byte 1, the last pair of numbers, specifies the width of the print fields. This is set to ØA to start with, which is hexadecimal for the decimal 1Ø.

Now, this all looks rather complicated but as soon as you come to use it you'll find it's not too bad at all. For example, if you want numbers with 2 decimal places PRINTed in fields 8 columns wide use @% = &2Ø2Ø8. If you want numbers PRINTed to one decimal place in fields 12 columns wide use @% = &2Ø1ØC, and so on.

Here's an example of why you might want to change the PRINT format. Type in and RUN this one-line program:

```
10 PRINT 1/2, 1/3, 1/4, 1/5, 1/6
```

Note that the numbers run into each other making them very difficult to read, and also the last two numbers are PRINTed on the line below. Now type @% = &2Ø4ØA and RUN the program again. This is a much neater display as the numbers are restricted to four decimal places. But one number still overflows the line. Now try @% = &2Ø4Ø8 and RUN it once more. The field width is reduced to 8 columns and all five figures fit on one line. The best way to get the hang of @% is to put in all sorts of different numbers (in hex of course) and then to try printing out lists of figures and words.

PRINT USING

Although screens can be formatted very well by utilising the PRINT@ command, the Dragon and Tandy have a very much more sophisticated family of PRINT commands—PRINT USING—which allow you far more control over your printed output, especially over numbers.

You can see most of the family of commands in action if you type in this program:

```
1Ø CLS
2Ø PRINT"□□THE DIRECTORS ARE
   PLEASED TO□□□□□□ ANNOUNCE
   THE PRELIMINARY□□□□□□□□
   □ACCOUNTS FOR 1984 OF:"
3Ø PRINT:PRINT"□□□the acme widget
   corporation"
4Ø PRINT:PRINT:PRINT"THE WORLD'S
   LEADING SUPPLIERS OF"
5Ø C$(Ø) = "blue□□□widgets":
   C$(1) = "green□□widgets":
   C$(2) = "yellow widgets":
   C$(3) = "red□□□□widgets"
6Ø FOR K = Ø TO 3
7Ø PRINT TAB(23 – LEN(C$(K)));C$(K)
8Ø NEXT
9Ø FOR K = 1TO7ØØØ:NEXT:CLS
11Ø PRINT@12,"JANUARY":
   PRINT@44,"———————"
12Ø PRINT:PRINT"AVERAGE WIDGET PRICES
   (WHOLESALE)"
13Ø A$ = "□□%□□□□%":B$ =
   "□□□□□□□□**$##.##"
14Ø PRINT:PRINTUSINGA$;C$(Ø);:
   PRINTUSINGB$;12.715265
15Ø PRINTUSINGA$;C$(1);:PRINT
   USINGB$;3.7363141
16Ø PRINTUSINGA$;C$(2);:PRINT
   USINGB$;1Ø.35824221
17Ø PRINTUSINGA$;C$(3);:PRINT
   USINGB$;.5163733
18Ø PRINT@416,USING"GROSS
   PROFIT□□□□$###,###
   .##";374241.5353
```

Lines 1Ø to 9Ø display a title page simply formatted with PRINT statements. Line 7Ø is interesting, though, because it shows how a series of strings can be centred on the screen with a number of TABs calculated according to the lengths of the strings.

The section from Line 1ØØ to Line 18Ø demonstrates the features of PRINT USING. Line 13Ø sets up A$ which will be used to display the first six letters of the strings defined in Line 5Ø. The length of the string is the number of spaces between the % signs, plus two—so the length of the string output includes spaces occupied by the % signs.

B$ deals with the format of the numerical output. The main function of numerical formatting is to tidy up a column of figures by making sure that they all contain the same number of digits, or are tabulated on the decimal point, or both.

The number of digits—in fact, the number of digits including the decimal point if there is one—is set by using the # symbol. Inserting a decimal point in the string of # marks will fix the position of the decimal point in each output. Look at Line 13Ø. B$ will PRINT numbers with two digits before, and two digits after the decimal point.

In addition, if you place a $ sign in front of the # signs you will have $ printed in front of the number. There is no £ sign for British users as the Dragon uses an American videochip.

If you specify ** before the # signs or $ sign, the leading spaces will be filled with asterisks. In the case of B$ all three have been used, but there's nothing to stop you using one, or any combination, of the formatting features, as your application demands.

Line 14Ø, then, will PRINT the first six characters of C$(Ø) followed by eight spaces and a $ sign, before 12.72—formatting numbers rounds the numbers either up or down.

Line 15Ø PRINTs the first six characters in A$, followed by *$3.74. Line 16Ø is similar to Line 14Ø, and finally Line 17Ø PRINTs the number as *$Ø.52.

If your numbers are very large, you have three choices of how they can be formatted. The simplest way is to have a long string of # signs. Or you can insert commas if you wish to indicate thousands. This is what you've done in Line 18Ø. The other alternative is to use exponential format. To see this in action alter Line 18Ø as follows:

```
18Ø PRINT@416,USING"GROSS PROFIT□
   □□□$##.##↑↑↑↑";374241.5353
```

The four ↑ signs mean 'PRINT in exponential format'—you must also limit the preceding numeric format to ##.##. One final interesting point about Line 18Ø is the combination of PRINT@ and PRINT USING. This allows you to PRINT formatted output at a specific point on the screen.

One feature of PRINT USING that hasn't been shown in the program is the ! sign. Try altering Line 13Ø so that it reads:

```
13Ø A$ = "□□!":B$ = "□□□□
   □□□□**$##.##"
```

When you RUN the program you'll see that ! tells the machine to PRINT the first character of the string only.

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 26...

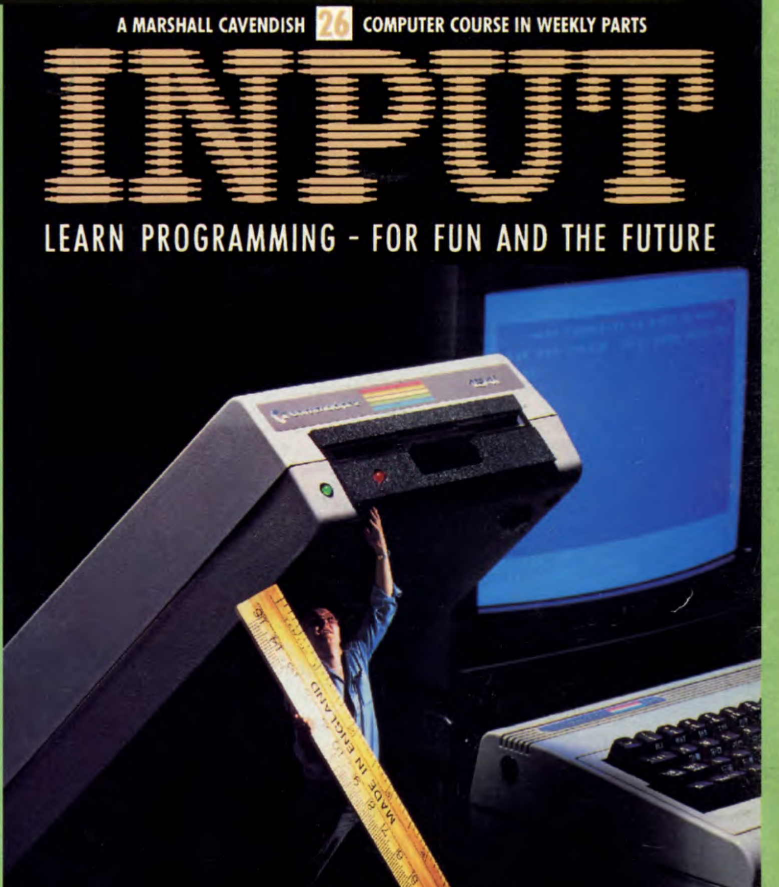
❑ *The number crunchers are out in force—a hungry snake digests the dwindling digits in the exciting SNAKE GAME*

❑ *Hit 'em right between the eyes with thrilling titles and prompts. Make the headlines really jump out of the screen with two ways to produce DISPLAY TYPEFACES on your micro*

❑ *Linking up the hardware is not as easy as it seems! So, in SETTING UP A DISK DRIVE, we explain the commands, procedures and pitfalls*

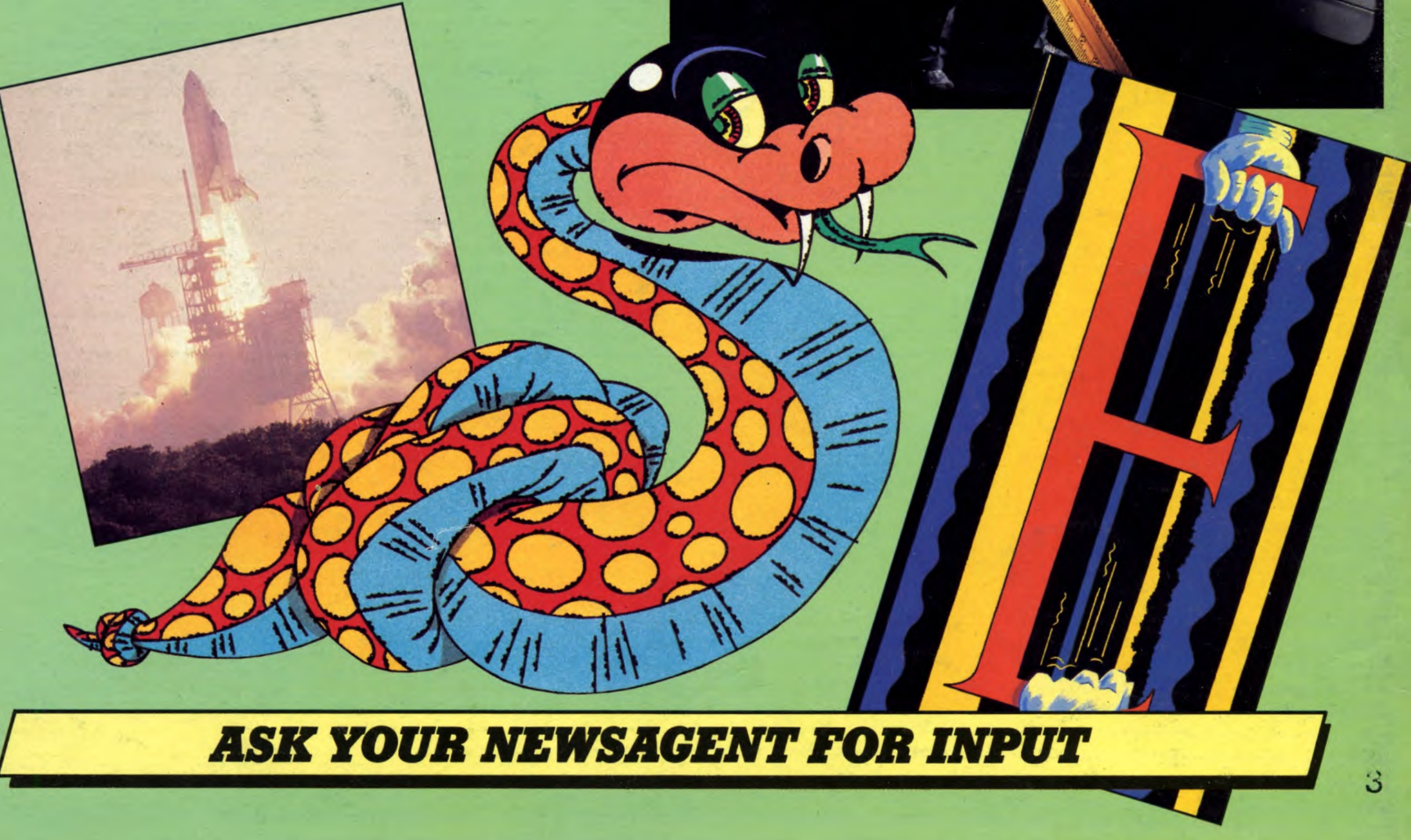
❑ *Muzzle velocity and elevation are crucial if you want to clear the barrier and trash the target with shots lobbed from your heavy howitzer in a TRAJECTORY GAME. Also find out about the realities of getting into orbit*

❑ *Plus a comprehensive 4 page INDEX to parts 14–26 of* INPUT


A MARSHALL CAVENDISH 26 COMPUTER COURSE IN WEEKLY PARTS

INPUT

LEARN PROGRAMMING - FOR FUN AND THE FUTURE



ASK YOUR NEWSAGENT FOR INPUT