A MARSHALL CAVENDISH 20 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 20**

INDEX

The last part of INPUT, Part 52, will contain a complete, cross-referenced index.

PICTURE CREDITS

Front cover, Ian Craig. Pages 605, 608, Trevor Lawrence/Projection. Pages 606, 607, Berry Fallon Design. Page 610, Trevor Lawrence. Page 611, Mickey Finn. Page 612, Chris Mynheer. Page 615, Jeremy Gower. Pages 616, 618, 620, Graeme Harris. Page 622, Kuo Kang Chen. Pages 624, 626, Ian Craig. Pages 628, 630, 633, David Lloyd.

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

There are four binders each holding 13 issues.

HOW TO ORDER YOUR BINDERS

UK and Republic of Ireland: Send £4.95 (inc p & p) (IR£5.95) for each binder to the address below:
Marshall Cavendish Services Ltd, Department 980, Newtown Road, Hove, Sussex BN3 7DN
Australia: See inserts for details, or write to INPUT, Times Consultants, PO Box 213, Alexandria, NSW 2015
New Zealand: See inserts for details, or write to INPUT, Gordon and Gotch (NZ) Ltd, PO Box 1595, Wellington
Malta: Binders are available from local newsagents.

BACK NUMBERS

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

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

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

COPIES BY POST

Our Subscription Department can supply copies to any UK address regularly at £1.00 each. For example the cost of 26 issues is £26.00; for any other quantity simply multiply the number of issues required by £1.00. Send your order, with payment to:
Subscription Department, Marshall Cavendish Services Ltd, Newtown Road, Hove, Sussex BN3 7DN
Please state the title of the publication and the part from which you wish to start.

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

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

INPUT IS SPECIALLY DESIGNED FOR:

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

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

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

GETTING THINGS IN PERSPECTIVE

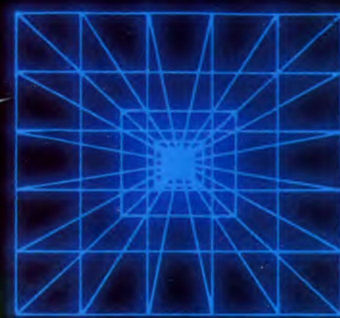
Impressive! That's the only way to describe the manipulative power of this program—the third in the series on wireframe graphics, that adds perspective and viewpoint

The wireframe drawing program given in the last two articles allows you to draw a cube in three dimensions and change its size. But it presents you with the same viewpoint every time; the front face of the cube is always facing the front, and although you can see through the wire frame, there's no way to view it from the top, side or anywhere else you fancy. This article gives you extra routines that allow you to specify the position of your eye so you can look at the cube from any direction.

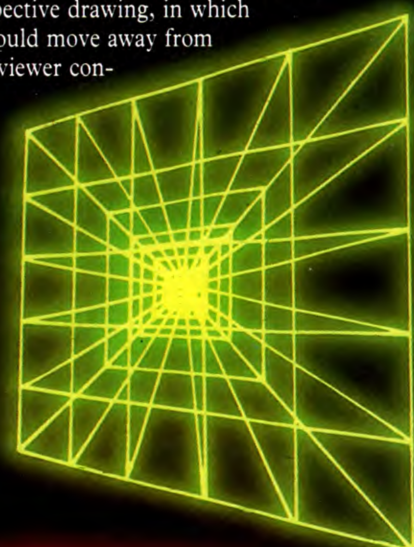
This is a very useful facility, especially when you come to draw more complicated objects, as the view from different directions may reveal hidden or obscured features. Also, by specifying a succession of different coordinates for the position of your eye, you can get an impression of 'flying by' the object, or walking round it. However, the sequence of drawings possible on a home computer is very slow and the effect doesn't match the speed of commercial wireframe drawings where the viewer seems to zoom round the car, planet or whatever at great speed. But the principles are the same.

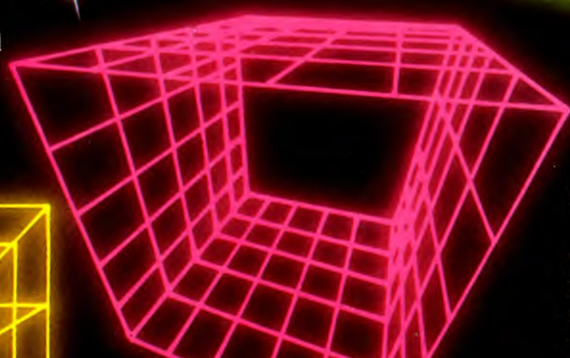
THE EFFECT OF VIEWPOINT

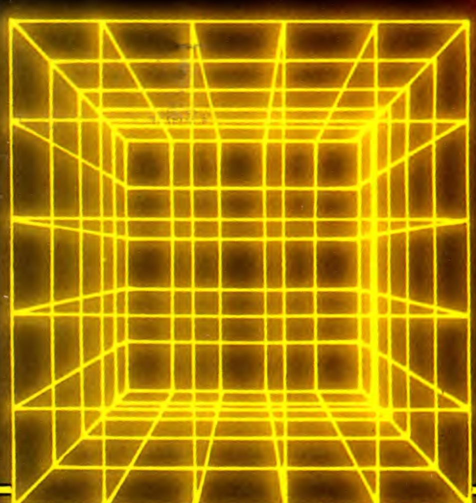
On page 560, we showed how it is possible to depict a three-dimensional object on a two-dimensional screen by using a visual convention which can be interpreted by the eye and brain of the viewer. The previous programs did this by using an isometric projection, in which slanting lines are understood to be receding or advancing from the screen.



The more common form of visual convention is perspective drawing, in which lines that would move away from the viewer converge towards a point in the distance, known as a vanishing point, and are foreshortened as they get 'further away'. In true three-point perspective, there are actually three vanishing points—one for lines drawn along each of the three axes. The position of the three points is fixed by the relationship between the position of the viewer and the position of the object to be depicted. (But remember, these points are imaginary within the visual convention.)







This gives a clue as to how an object like a cube can be depicted from different viewpoints in three-dimensional space. It will be necessary to set up further transformations which produce the effect of convergence towards a vanishing point, and also foreshorten the object.

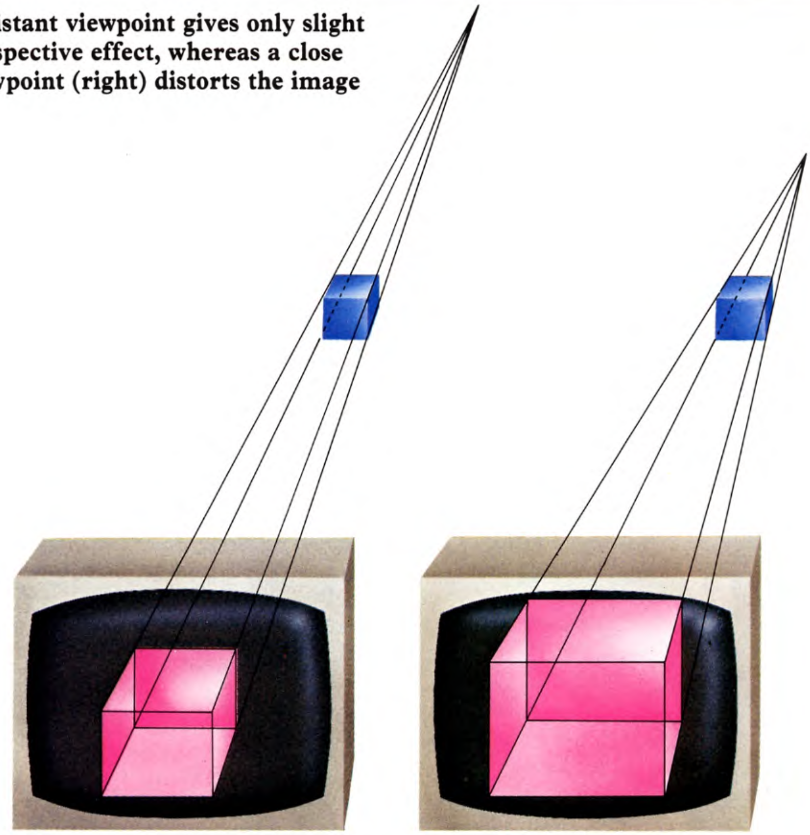
The first thing you will need to know is the relationship between the position of the viewer and that of the object. To determine this, you have to relate the viewpoint to the X, Y and Z axes which you have set up on the screen. The programs which follow do just that, and then perform the necessary transformations.

BEFORE YOU START

To RUN the new sections of program listed here you need the Grid routine given on page 512 and the Acorns need the Line routine too (page 511). So, if you SAVEd a copy of those routines you should LOAD them in now. It's a good idea to LOAD in the Circle drawing routine on page 513 as well. You won't need it this time but it is handy to have all the routines together ready for the globe drawing program which follows in the next article. So the lines you need are 5ØØØ to 616Ø (and 95ØØ to 955Ø for the Acorn as well); all other lines can be deleted.

As before, you'll have to type in several sections of program before you can RUN anything—because the eye position as well as the transformations for rotation and perspective have to be set up first.

A distant viewpoint gives only slight perspective effect, whereas a close viewpoint (right) distorts the image

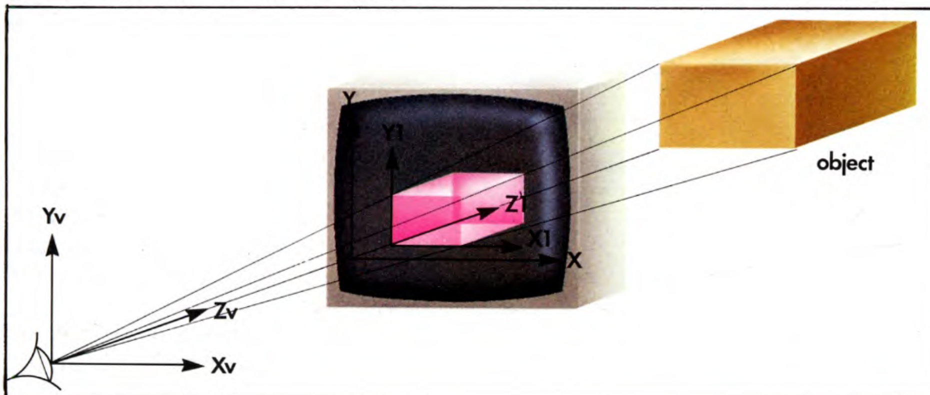


DEFINING YOUR VIEWPOINT

The first section of program calculates the variables needed to determine the position of your eye in three-dimensional space:

```
8ØØØ LET XV=X: LET YV=Y: LET ZV=Z
8Ø1Ø LET WV=YV*YV+ZV*ZV
8Ø2Ø LET PV=SQR (XV*XV+WV)
8Ø3Ø IF PV=Ø THEN RETURN
8Ø35 LET WV=SQR WV
8Ø4Ø LET XU=XV/PV
8Ø5Ø LET YU=YV/PV
8Ø6Ø LET ZU=ZV/PV
8Ø7Ø LET WU=WV/PV
8Ø8Ø REM EYE-ORIENTATION
8Ø9Ø LET A=XV*YV: LET B=ZV: GOSUB
   845Ø: LET G=H
81ØØ LET A=YV: LET B=XV: GOSUB 845Ø:
   LET G=G+H
811Ø LET SG=SIN G
812Ø LET CG=COS G
814Ø LET R1=WU*CG
815Ø LET R2=-WU*SG
816Ø LET R3=-XU
817Ø LET R4=-YU
818Ø LET R5=-ZU
819Ø LET R6=XV*XU+YV*YU+ZV*ZU
82ØØ IF WU=Ø THEN GOTO 835Ø
821Ø LET XT=XV*WU-(YV*YU
   +ZV*ZU)*XU/WU
822Ø LET YT=(YV*ZU-ZV*YU)/WU
823Ø LET R7=(ZU*SG-XU*YU*CG)/WU
824Ø LET R8=(-YU*SG-XU*ZU*CG)/WU
825Ø LET R9=CG*XT+SG*YT
826Ø LET S1=(ZU*CG+XU*YU*SG)/WU
827Ø LET S2=(-YU*CG+XU*ZU*SG)/WU
828Ø LET S3=-SG*XT+CG*YT
833Ø RETURN
835Ø LET R7=-1
836Ø LET R8=Ø
837Ø LET R9=Ø
838Ø LET S1=Ø
839Ø LET S2=1
84ØØ LET S3=Ø
841Ø RETURN
845Ø IF B<>Ø THEN LET H=ATN (A/B):
   RETURN
846Ø LET H=PI/2: RETURN
```

To represent a three-dimensional image coordinates are transformed from three-dimensional axes (X1, Y1, Z1) and (Xv, Yv, Zv) to the screen axes (X,Y)



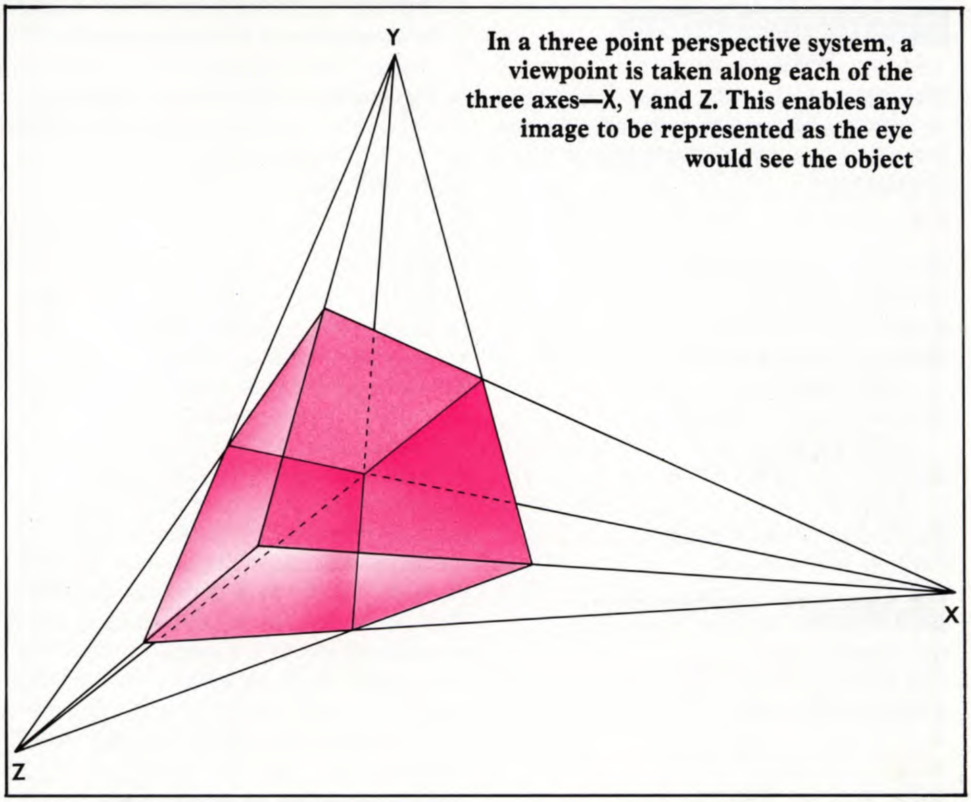

```
8ØØØ XV=X:YV=Y:ZV=Z
8Ø1Ø WV=YV*YV+ZV*ZV
8Ø2Ø PV=SQR(XV*XV+WV)
8Ø3Ø IF PV=Ø THEN RETURN
8Ø35 WV=SQR(WV)
8Ø4Ø XU=XV/PV
8Ø5Ø YU=YV/PV
8Ø6Ø ZU=ZV/PV
8Ø7Ø WU=WV/PV
8Ø8Ø REM EYE-ORIENTATION
8Ø9Ø A=XV*YV:B=ZV:GOSUB 845Ø:
   G=H
81ØØ A=YV:B=XV:GOSUB 845Ø:
   G=G+H
```

```
811Ø SG=SIN(G)
812Ø CG=COS(G)
813Ø REM ROTATION MATRIX
814Ø R1=WU*CG
815Ø R2=-WU*SG
816Ø R3=-XU
817Ø R4=-YU
818Ø R5=-ZU
819Ø R6=XV*XU+YV*YU+ZV*ZU
82ØØ IF WU=Ø THEN 834Ø
821Ø XT=XV*WU-(YV*YU+ZV*ZU)
    *XU/WU
822Ø YT=(YV*ZU-ZV*YU)/WU
823Ø R7=(ZU*SG-XU*YU*CG)/WU
824Ø R8=(-YU*SG-XU*ZU*CG)/WU
825Ø R9=CG*XT+SG*YT
826Ø S1=(ZU*CG+XU*YU*SG)/WU
827Ø S2=(-YU*CG+XU*ZU*SG)/WU
828Ø S3=-SG*XT+CG*YT
833Ø RETURN
834Ø REM SPECIAL CASE ON X-AXIS
835Ø R7=-1
836Ø R8=Ø
837Ø R9=Ø
838Ø S1=Ø
839Ø S2=1
84ØØ S3=Ø
841Ø RETURN
845Ø IF B<>Ø THEN H=ATN(A/B):
    RETURN
846Ø H=π/2:RETURN
```

```
8ØØØ DEF PROCposition(X,Y,Z)
8Ø1Ø LOCAL WV,WU,G,SG,CG,XT,YT
8Ø2Ø XV=X
8Ø3Ø YV=Y
8Ø4Ø ZV=Z
8Ø5Ø WV=YV*YV+ZV*ZV
8Ø6Ø PV=SQR(XV*XV+WV)
8Ø7Ø WV=SQR(WV)
8Ø8Ø IF PV=Ø THEN ENDPROC
8Ø9Ø XU=XV/PV
81ØØ YU=YV/PV
811Ø ZU=ZV/PV
812Ø WU=WV/PV
813Ø REM EYE ORIENTATION
814Ø G=FNatan(XV*YV,ZV)
    +FNatan(YV,XV)
815Ø SG=SIN(G)
816Ø CG=COS(G)
817Ø REM ROTATION MATRIX
818Ø R11=WU*CG
819Ø R21=-WU*SG
82ØØ R31=-XU
821Ø R32=-YU
822Ø R33=-ZU
823Ø R34=XV*XU+YV*YU+ZV*ZU
824Ø IF WU=Ø THEN 834Ø
825Ø XT=XV*WU-(YV*YU+ZV*ZU)
    *XU/WU
826Ø YT=(YV*ZU-ZV*YU)/WU
827Ø R12=(ZU*SG-XU*YU*CG)/WU
828Ø R13=(-YU*SG-XU*ZU*CG)/WU
829Ø R14=CG*XT+SG*YT
83ØØ R22=(ZU*CG+XU*YU*SG)/WU
831Ø R23=(-YU*CG+XU*ZU*SG)/WU
832Ø R24=-SG*XT+CG*YT
833Ø ENDPROC
834Ø REM SPECIAL CASE ON X-AXIS
835Ø R12=-1
836Ø R13=Ø
837Ø R14=Ø
838Ø R22=Ø
839Ø R23=1
84ØØ R24=Ø
841Ø ENDPROC
845Ø DEF FNatan(A,B)
846Ø IF B<>Ø THEN =ATN(A/B) ELSE=PI/2
```

Key in Lines 8ØØØ to 841Ø as for the Commodores, then add the following lines:

```
845Ø IF B<>Ø THEN H=ATN(A/B) ELSE
    H=PI/2
846Ø RETURN
```



In a three point perspective system, a viewpoint is taken along each of the three axes—X, Y and Z. This enables any image to be represented as the eye would see the object

To understand what's going on you have to remember that the Z axis is the one coming towards you out of the centre of the screen, the Y axis points up the screen, and the X axis points along the screen.

The position of the eye is at (XV, YV, ZV); the V stands for Viewpoint. The variable WV gives the distance in the Y and Z directions combined. The position of the object you're looking at is assumed to be the origin (Ø, Ø, Ø) in space which is placed at the centre of the screen for simplicity. Line 8Ø3Ø (8Ø8Ø for Acorn) causes the routine to abort if the eye position is placed at the origin, because it is difficult to look at your own eye—without a mirror. The variables XU, YU and ZU are the distances in the X, Y and Z axis directions of a line of unit length drawn between the origin and the eye position. WU is the distance in the Y and Z directions combined.

When viewing an object, you can get different views by moving around the object from left or right. This angle is measured starting from the X axis and is set at Line 8Ø9Ø (814Ø for Acorn) and 81ØØ. A check is made (Lines 845Ø and 846Ø) to prevent division by zero, which would interrupt the program by causing an error message.

This gives a normal view when you are looking directly along the X axis and it gradually rotates round as you move round, giving more interesting views. Lines 814Ø to 828Ø (818Ø to 832Ø for Acorns) set variables that define the orientation of the eye position in space. The eye position is located so that its Z axis lies along the line from the eye to the origin of the screen.

The rest of the routine sets variables for the special case, when the eye position is actually on the X axis.

THE TRANSFORMATIONS

The next section transforms the X, Y and Z coordinates of the cube to the final screen coordinates. These transformations take into account the position of the eye and the effect of perspective:

```
8500 LET X1=T1*X+T4*Y+T7
8510 LET Y1=T2*X+T5*Y+T8
8520 LET Z1=T3*X+T6*Y+T9
8540 LET X2=R1*X1+R7*Y1
    +R8*Z1+R9
8550 LET Y2=R2*X1+S1*Y1
    +S2*Z1+S3
8560 LET Z2=R3*X1+R4*Y1
    +R5*Z1+R6
8575 IF Z2<ZN THEN RETURN
8580 LET X3=D*X2/Z2
8590 LET Y3=-D*Y2/Z2
8600 RETURN
```

```
8500 X1=T1*X+T4*Y+T7
8510 Y1=T2*X+T5*Y+T8
8520 Z1=T3*X+T6*Y+T9
8530 REM 2-D TO 3-D
8540 X2=R1*X1+R7*Y1+R8*Z1+R9
8550 Y2=R2*X1+S1*Y1+S2*Z1+S3
8560 Z2=R3*X1+R4*Y1+R5*Z1+R6
8570 REM OBJECT TO EYE
8575 IF Z2<ZN THEN RETURN
8580 X3=D*X2/Z2
8590 Y3=D*Y2/Z2
8600 RETURN
```

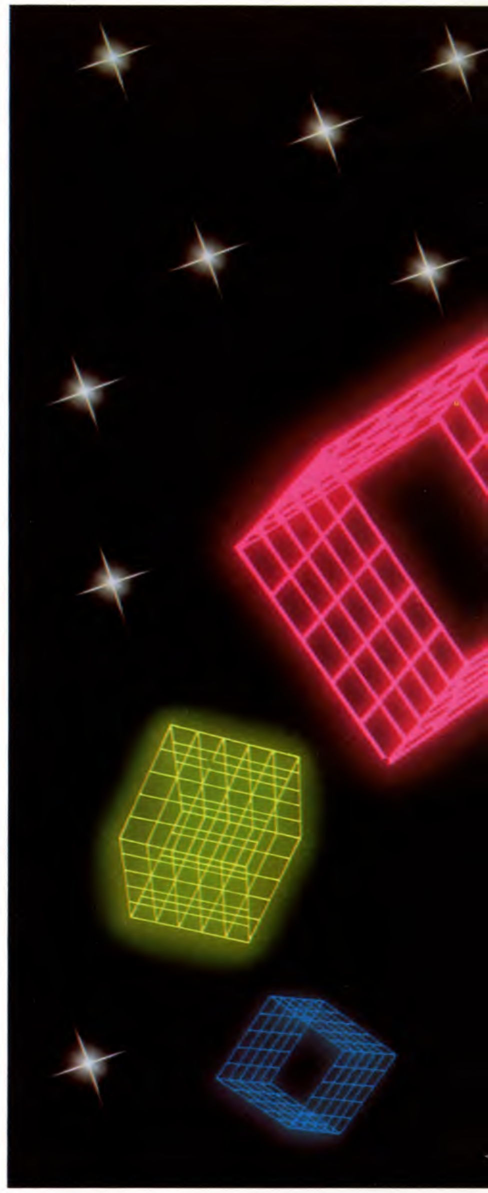
```
8500 DEF FNtrans(X,Y)
8510 LOCAL X1,Y1,Z1,X2,Y2,Z2
8520 REM 2-D TO 3-D
8530 X1=T11*X+T12*Y+T13
8540 Y1=T21*X+T22*Y+T23
8550 Z1=T31*X+T32*Y+T33
8560 REM OBJECT TO EYE
8570 X2=R11*X1+R12*Y1
    +R13*Z1+R14
8580 Y2=R21*X1+R22*Y1
    +R23*Z1+R24
8590 Z2=R31*X1+R32*Y1
    +R33*Z1+R34
8600 IF Z2<ZMIN THEN =FALSE
8610 REM 3-D TO 2-D
8620 X3=D*X2/Z2
8630 Y3=D*Y2/Z2
8640 =TRUE
```

This routine uses complicated matrix arithmetic to transform the coordinates (X,Y), but basically they are the transformation steps described on pages 561 to 564. Lines 8500 to 8520 (8530 to 8550 for Acorn) transform the 2-D plotting plane (X,Y) to 3-D space (X1, Y1, Z1). Lines 8540 to 8560 (8570 to 8590 for Acorn) transform the 3-D space coordinates (X1, Y1, Z1) to be positioned according to the eye position and direction (X2, Y2, Z2). Line 8575 (8600 for Acorn) checks whether the new position to be plotted is too close to the eye position—that is, Z2 lies between 0 and ZN (or 0 and ZMIN on the Acorns), or whether it is behind the eye (Z2 negative). In both these cases, the plotting position is ignored as it would be impossible to view the object. If the position is farther away than ZN or ZMIN from the front of the eye position, then the screen coordinate position (X3, Y3) is calculated—at the end of the routine. The D/Z2 parameter on these lines adds perspective to the picture by projecting the object on to a flat screen at a set distance D from the eye. If you set D to a small value this has the effect of moving the screen close to the eye so the perspective is more pronounced. When D is large, the screen is a long way off, so the perspective effect is quite small—the image appears virtually undistorted, even when it is viewed obliquely in any direction.

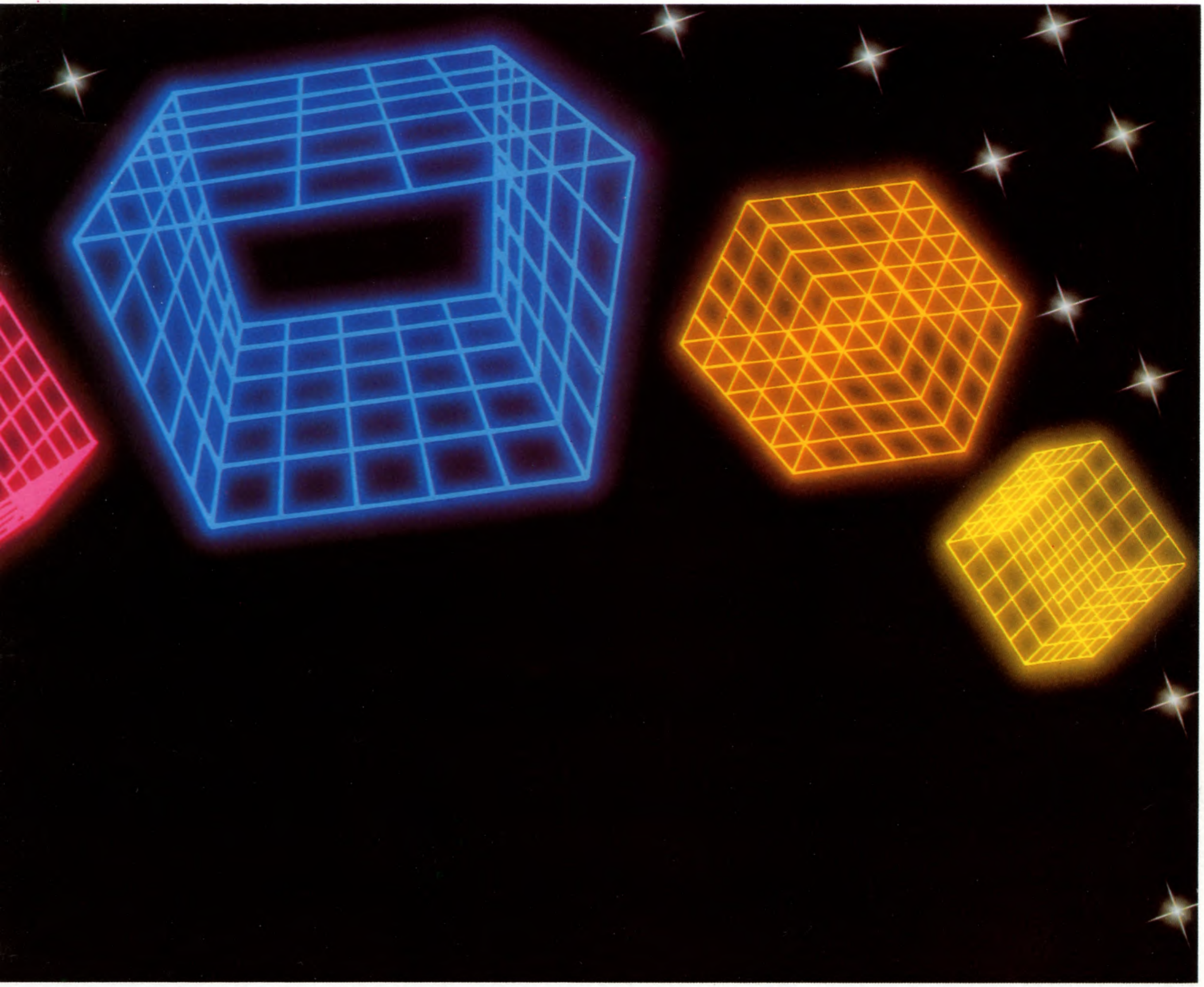
SETTING INITIAL VARIABLES

The next step is to rewrite the Initialize and Draw routines to make use of the new Transform routines:

```
9000 CLS
9020 LET XM=256: LET YM=176
9030 LET XD=XM/2: LET YD=YM/2
9040 LET ZN=1
9042 INPUT "ENTER PROJECTION PLANE
    DISTANCE",D
9045 IF D<=0 THEN LET D=1000*ZN
9050 LET T1=1: LET T2=0: LET T3=0
9060 LET T4=0: LET T5=1: LET T6=0
9070 LET T7=0: LET T8=0: LET T9=0
9090 CLS : RETURN
9500 LET X=XS: LET Y=YS:
    GOSUB 8500: IF Z2<ZN THEN
    GOTO 9520
9505 IF X3<-127 OR Y3<-87 OR
    X3>128 OR Y3>88 THEN
    GOTO 9550
9510 PLOT 127+X3,87+Y3
9520 LET X=XE: LET Y=YE:
    GOSUB 8500: IF Z2<Zn THEN
    GOTO 9550
9525 IF X3<-127 OR Y3<-87 OR
    X3>128 OR Y3>88 THEN
    GOTO 9550
9530 DRAW 127+X3-PEEK 23677,
    87+Y3-PEEK 23678
9550 RETURN
```



```
9000 PRINT "♥"
9020 XM=320:YM=200
9030 XD=XM/2:YD=YM/2
9040 ZN=1
9042 INPUT "ENTER PROJECTION PLANE
    DISTANCE";D
9045 IF D=<0 THEN D=1000*ZN
9050 T1=0:T2=0:T3=0
9060 T4=0:T5=0:T6=0
9070 T7=0: T8=0:T9=0
9085 PRINT "♥"
9090 RETURN
9500 X=XS:Y=YS:GOSUB 8500:
    IF Z2<ZN THEN 9520
9505 IF X3<-159ORY3<-99ORX3
    >159ORY3>99 THEN 9550
9510 IX=INT(160+X3):
    IY=(100-Y3)
```



```
9520 X=XE:Y=YE:GOSUB 8500:
   IF Z2<ZN THEN 9550
9525 IF X3<-159ORY3<-99ORX3
   >159ORY3>99 THEN 9550
9540 LINE IX,IY,160+X3,
   100-Y3,1
9550 RETURN
```

```
9000 SCNCLR
9020 XM=1023:YM=1023
9030 XD=XM/2:YD=YM/2
9040 ZN=1
9042 PRINT "♥ENTER PROJECTION
   PLANE DISTANCE":INPUT D
9045 IF D=<0 THEN D=
   1000*ZN
9050 T1=0:T2=0:T3=0
9060 T4=0:T5=0:T6=0
9070 T7=0:T8=0:T9=0
9085 PRINT "♥"
9090 RETURN
9500 X=XS:Y=YS:GOSUB 8500:
   IF Z2<ZN THEN 9520
9505 IF X3<-511ORY3<-511ORX3
   >511ORY3>511 THEN 9550
9510 POINT 2,INT(511+X3),
   INT(511-Y3)
9520 X=XE:Y=YE:GOSUB 8500:
   IF Z2<ZN THEN 9550
9525 IF X3<-511ORY3<-511ORX3
   >511ORY3>511 THEN 9550
9530 DRAW 2 TO 511+X3,511-Y3
9550 RETURN
```

```
9000 DEF PROCINIT
9010 CLS:CLG
9020 XMAX=1280:YMAX=1024
9030 XMID=XMAX/2:YMID=YMAX/2
9035 VDU29,XMID;YMID;
9040 ZMIN=1
9042 INPUT"ENTER PROJECTION PLANE
   DISTANCE□",D
9045 IF D<=0 THEN D=1000*ZMIN
9050 PROCXvector(1,0,0)
9060 PROCYvector(0,1,0)
9070 PROCorigin(0,0,0)
9085 CLS
9090 ENDPROC
9100 DEF PROCMOVE(X,Y)
9120 IF FNtrans(X,Y) THEN MOVE X3,Y3
9130 ENDPROC
9200 DEF PROCDRAW(X,Y)
9220 IF FNtrans(X,Y) THEN
   DRAW X3,Y3
9230 ENDPROC
```

```
9000 PCLS
9020 XM=256:YM=192
9030 XD=XM/2:YD=YM/2
9040 ZN=1
9042 CLS:INPUT"□ENTER PROJECTION
   PLANE DISTANCE□";D
9045 IF D=<0 THEN D=1000*ZN
9050 T1=1:T2=0:T3=0
9060 T4=0:T5=1:T6=0
9070 T7=0:T8=0:T9=0
9085 CLS
9090 RETURN
9500 X=XS:Y=YS:GOSUB8500:
   IF Z2<ZN THEN 9520
9505 IFX3<-127ORY3<-96ORX3
   >128ORY3>95 THEN 9550
9510 DRAW"BM"+STR$(INT(127
   +X3))+","+STR$(INT(95-Y3))
9520 X=XE:Y=YE:GOSUB8500:
   IF Z2<ZN THEN 9550
9525 IFX3<-127ORY3<-96ORX3
   >128ORY3>95 THEN 9550
9530 LINE-(127+X3,95-Y3),PSET
9550 RETURN
```

Line 9020 sets the maximum dimensions of the screen in the X and Y directions, and Line 9030 sets the mid point. Line 9040 sets the closest allowable position of points to be plotted to the eye position. The variable D gives the actual distance from the eye position to the projection plane—the screen—and determines the perspective. Values of D are entered when you RUN the program, so you can vary the degree of perspective. If no value is given—by pressing ENTER or RETURN—then Line 9045 sets a default value of 1000. Lines 9050 to 9070 pass values to the transformation constants, to specify the plane in which the image is to be drawn in three-dimensional space.

The section of program above continues with the revised drawing routine. For Acorn micros, Line 9120 MOVEs the cursor to the required screen position, and Line 9220 DRAWs to a new position. For the other computers, however, variables for the transformation constants are first set (Line 9500), then a check is made (Lines 9505 and 9525) to determine whether a new point to be plotted (Lines 9510 and 9530) is on the screen. This check is necessary to prevent an error message if you try to plot off the screen.

CALLING THE ROUTINE

You now need the routine to draw the grid. If you do not have a typed copy of the program from the previous article, then enter those program lines now, as well as this program.

```
110 GOSUB 9000
120 LET L=20: LET N=3
125 GOSUB 505: GOTO 140
130 GOSUB 500
140 IF X=0 AND Y=0 AND Z=0 THEN
   GOTO 170
150 GOSUB 1000
160 GOTO 130
170 CLS
180 STOP
500 IF INKEY$="" THEN GOTO 500
505 CLS
510 INPUT "INPUT EYE POSITION
   (X,Y,Z)",X,Y,Z
520 GOSUB 8000
530 RETURN
1000 LET P=L/2
1010 LET T1=1: LET T2=0: LET T3=0
1020 LET T4=0: LET T5=1: LET T6=0
1030 LET T7=-P: LET T8=-P: LET
   T9=-P
1040 GOSUB 1200: REM BOTTOM
1050 LET T7=-P: LET T8=-P:
   LET T9=P
1060 GOSUB 1200: REM TOP
1070 LET T4=0: LET T5=0: LET T6=-1
1080 GOSUB 1200: REM LEFT
1090 LET T7=-P: LET T8=P: LET T9=P
1100 GOSUB 1200: REM RIGHT
1110 LET T1=0: LET T2=-1: LET T3=0
1120 GOSUB 1200: REM BACK
1130 LET T7=P: LET T8=P: LET T9=P
1140 GOSUB 1200: REM FRONT
1170 RETURN
1200 LET XA=0: LET YA=0: LET LW=L:
   LET LH=L: LET NX=N: LET NY=N
1210 GOSUB 5000
1220 RETURN
```

```
100 PI=π
110 GOSUB 9000
120 L=100:N=3
125 GOSUB 505:GOTO 140
130 GOSUB 500
140 IF X=0ANDY=0ANDZ=0THEN170
150 GOSUB1000
160 GOTO 130
170 NRM:PRINT "♥":END
500 IF PEEK(197)=64 THEN 500
505 NRM
510 INPUT"♥INPUT EYE POSITION
   (X,Y,Z)";X,Y,Z
520 GOSUB 8000
530 HIRES 0,1
540 RETURN
1000 P=L/2
1010 T1=1:T2=0:T3=0
1020 T4=0:T5=1:T6=0
1030 T7=-P:T8=-P:T9=-P
1040 GOSUB 1200
1050 T7=-P:T8=-P:T9=P
1060 GOSUB 1200
1070 T4=0:T5=0:T6=-1
1080 GOSUB 1200
1090 T7=-P:T8=P:T9=P
1100 GOSUB 1200
1110 T1=0:T2=-1:T3=0
1120 GOSUB 1200
1130 T7=P:T8=P:T9=P
1140 GOSUB 1200
1170 RETURN
1200 XA=0:YA=0:LW=L:LH=L:
   NX=N:NY=N
1210 GOSUB 5000
1220 RETURN
```

```
100 PI=π
110 GOSUB 9000
120 L=100:N=3
125 GOSUB 505:GOTO 140
130 GOSUB 500
140 IF X=0ANDY=0ANDZ=0THEN170
150 GOSUB1000
160 GOTO 130
170 GRAPHIC 0:PRINT"♥":END
500 IF PEEK(197)=64 THEN 500
505 GRAPHIC 0
510 PRINT"♥INPUT EYE POSITION":
   INPUT"(X,Y,Z)";X,Y,Z
520 GOSUB 8000
530 GRAPHIC 2
540 RETURN
1000 P=L/2
1010 T1=1:T2=0:T3=0
1020 T4=0:T5=1:T6=0
1030 T7=-P:T8=-P:T9=-P
1040 GOSUB 1200
1050 T7=-P:T8=-P:T9=P
1060 GOSUB 1200
1070 T4=0:T5=0:T6=-1
1080 GOSUB 1200
1090 T7=-P:T8=P:T9=P
1100 GOSUB 1200
1110 T1=0:T2=-1:T3=0
1120 GOSUB 1200
1130 T7=P:T8=P:T9=P
1140 GOSUB 1200
1170 RETURN
1200 XA=0:YA=0:LW=L:LH=L:
   NX=N:NY=N
1210 GOSUB5000
1220 RETURN
```

```
100 MODE0
110 PROCINIT
120 L=100:N=5
130 REPEAT
```

```
140 V = FNgetposition
150 IF V THEN PROCCUBE
160 UNTIL NOT V
170 MODE1
180 END
5ØØ DEF FNgetposition
51Ø INPUT"EYE POSITION (X,Y,Z)□",
   X,Y,Z
52Ø PROCposition(X,Y,Z)
53Ø CLS
54Ø =(X<>Ø)OR(Y<>Ø)OR(Z<>Ø)
1ØØØ DEF PROCCUBE
1Ø1Ø LOCAL P
1Ø2Ø P=L/2
1Ø3Ø PROCXvector(1,Ø,Ø)
1Ø4Ø PROCYvector(Ø,1,Ø)
1Ø5Ø PROCorigin(-P,-P,-P)
1Ø6Ø PROCSIDE:REM BOTTOM
1Ø7Ø PROCorigin(-P,-P,P)
1Ø8Ø PROCSIDE:REM TOP
1Ø9Ø PROCYvector(Ø,Ø,-1)
11ØØ PROCSIDE:REM LEFT
111Ø PROCorigin(-P,P,P)
112Ø PROCSIDE:REM RIGHT
113Ø PROCXvector(Ø,-1,Ø)
114Ø PROCSIDE:REM BACK
115Ø PROCorigin(P,P,P)
116Ø PROCSIDE:REM FRONT
117Ø ENDPROC
12ØØ DEF PROCSIDE
121Ø PROCGRID(Ø,Ø,L,L,N,N)
122Ø ENDPROC
96ØØ DEF PROCXvector(DX,DY,DZ)
961Ø T11=DX
962Ø T21=DY
963Ø T31=DZ
964Ø ENDPROC
97ØØ DEF PROCYvector(DX,DY,DZ)
971Ø T12=DX
972Ø T22=DY
973Ø T32=DZ
974Ø ENDPROC
98ØØ DEF PROCorigin(X,Y,Z)
981Ø T13=X
982Ø T23=Y
983Ø T33=Z
984Ø ENDPROC
```

```
1ØØ PI=4*ATN(1):PMODE4,1
11Ø GOSUB 9ØØØ
12Ø L=2Ø:N=5
125 GOSUB5Ø5:GOTO 14Ø
13Ø GOSUB 5ØØ
14Ø IF X=ØANDY=ØANDZ=Ø
   THEN17Ø
15Ø GOSUB 1ØØØ
16Ø GOTO 13Ø
17Ø CLS
18Ø END
5ØØ IF INKEY$="" THEN 5ØØ
5Ø5 CLS
51Ø INPUT"□INPUT EYE POSITION
   (X,Y,Z)□□□";X,Y,Z
52Ø GOSUB8ØØØ
53Ø PCLS:SCREEN1,1
54Ø RETURN
1ØØØ P=L/2
1Ø1Ø T1=1:T2=Ø:T3=Ø
1Ø2Ø T4=Ø:T5=1:T6=Ø
1Ø3Ø T7=-P:T8=-P:T9=-P
1Ø4Ø GOSUB 12ØØ 'BOTTOM
1Ø5Ø T7=-P:T8=-P:T9=P
1Ø6Ø GOSUB 12ØØ 'TOP
1Ø7Ø T4=Ø:T5=Ø:T6=-1
1Ø8Ø GOSUB 12ØØ 'LEFT
1Ø9Ø T7=-P:T8=P:T9=P
11ØØ GOSUB 12ØØ 'RIGHT
111Ø T1=Ø:T2=-1:T3=Ø
112Ø GOSUB 12ØØ 'BACK
113Ø T7=P:T8=P:T9=P
114Ø GOSUB 12ØØ 'FRONT
117Ø RETURN
12ØØ XA=Ø:YA=Ø:LW=L:LH=L:
   NX=N:NY=N
121Ø GOSUB 5ØØØ
122Ø RETURN
```

Now RUN the program. If the program is working correctly, it enters the 'Initialization' routine starting at Line 11Ø. This routine sets up some variables and prints a prompt for you to enter a value for D—the projection plane distance. The perspective built into the program is such that the greater this distance, the farther away the object appears which means it shows very little perspective. Enter a value of 1ØØØ to begin with. The program returns to Line 12Ø, which specified the length L of each side of the cube, and the number of grid squares N along each side. Lines 13Ø to 16Ø read in the coordinates of the eye position (Line 51Ø), and then draw the cube from that position. As soon as the first view is drawn you can enter a new set of coordinates to see the cube from a new direction. When you enter the values Ø, Ø and Ø, the program ends. The routine from Lines 5ØØ to 53Ø actually calls the Position routine (Line 52Ø) to set up the transformation constants. The Cube routine (Lines 1ØØØ to 117Ø) then positions and draws each of the six sides. The Side routine (Lines 12ØØ to 122Ø) plots each side as a grid, using the Grid routine.

Try different values for the eye position to see the effect on the view. A value of 1ØØØ for D and 2ØØ, Ø and Ø for X, Y and Z are good to start with. Then try 1ØØ for D and 2Ø, Ø, Ø for X, Y, and Z. The cube appears the same size but the perspective is much more pronounced. To enter new values for X, Y and Z, press any key and the prompt will appear. To enter new values for D you have to press BREAK and then RUN the program again. You'll be given a prompt for D then a second prompt for X, Y and Z.

On machines like the Acorn, the view is automatically 'clipped' if any part of it falls outside the screen. Thus it does not matter if you come in really close and the cube spills off the edge of the screen. This can give more spectacular views with exaggerated perspective. On some machines such as the Spectrum, an error is reported if points to be plotted fall outside the screen area. Thus the program ignores any line which would go out of the available area. This may lead to odd results since all of a line is omitted, even if just the tip of it would be over the maximum permitted point. Points closer than a certain minimum distance, and points behind the eye, ZN(ZMIN for Acorns), are ignored. So if the eye position is very close to or inside the cube, spurious results will also occur. It is possible to check for such points and clip the lines approximately but this requires considerable extra code and computation.

Save a copy of the complete listing on tape or disk, because in a future article you can learn how to use the same routines to draw duplicate shapes, and to produce some spectacular circular graphics.

MODEMS-YOUR LINK TO THE WORLD

If you're fed up with games or simply want to use your computer to its fullest, why not consider linking yourself to bigger and better computers?

Even the humblest of home computers can be connected to the telephone system and thence to some of the most powerful computers and biggest databases on Earth. Linked in this way you can access phenomenal amounts of information or, in a more practical sense, communicate with other computer enthusiasts locally, nationally, or internationally.

'Hacking' is the popular term used to describe computer use of this sort. And while it seems improbable that you could infiltrate sensitive databases (though this has happened!) or trigger the next World War (which has only happened in fiction), almost anything is possible. A computer, a linking device called a *modem*, a telephone and the appropriate, usually very simple, software can quite literally open up the world.

Computers can communicate with each other in three basic ways. They can send messages to each other over long distances using telephone lines or radio waves (or in some cases both). They can share the same information storage system (which in practice means sharing the same disk units). Or they can be connected directly and share each other's processor and memory.

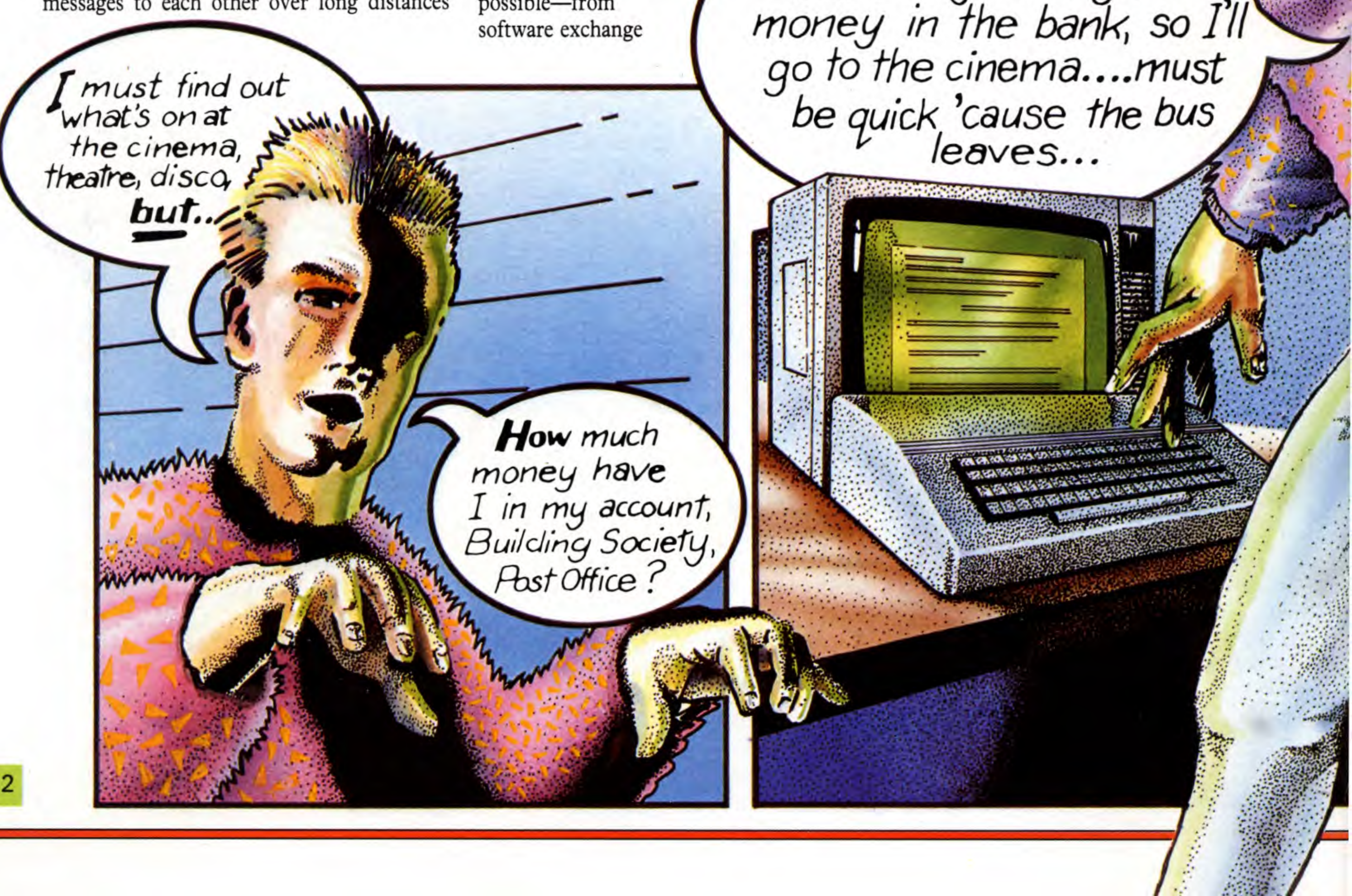
HELLO WORLD

Of these options, the first is undoubtedly the most exciting, even thrilling, step forward for the home computing enthusiast. It is already possible to obtain equipment and software, at reasonable prices, which will enable your computer to talk to another computer almost anywhere on the globe over the very same 'lines' which are used already for telecommunications.

Through the telephone system, access to a huge range of facilities is possible—from software exchange set up amongst groups of friends to huge business, news and information services costing users several thousand pounds a year.

Computer communications is one of those areas of new technology where the future really is here today. By enabling one computer to talk to another over long distances it is already possible to live and work from your own home without ever stepping outside the front door.

You can peruse shopping lists, examine illustrations of the goods for sale, compare prices and order what you want from the comfort of your own armchair. You can control your finances, pay bills instantly and



keep an up-to-the-minute check on your financial position. An increasing number of jobs can be done from home using a micro, especially those office jobs which involve using and processing information.

A secretary and boss, for example, could be miles apart in their own homes and still work well together using computer communications. Having roughed out a letter on his or her own micro the boss can send it, complete with spelling mistakes and bad grammar, to the secretary who then corrects it, formats it so that it looks presentable, lets the boss have a quick check and, finally, sends it.

BULLETIN BOARDS

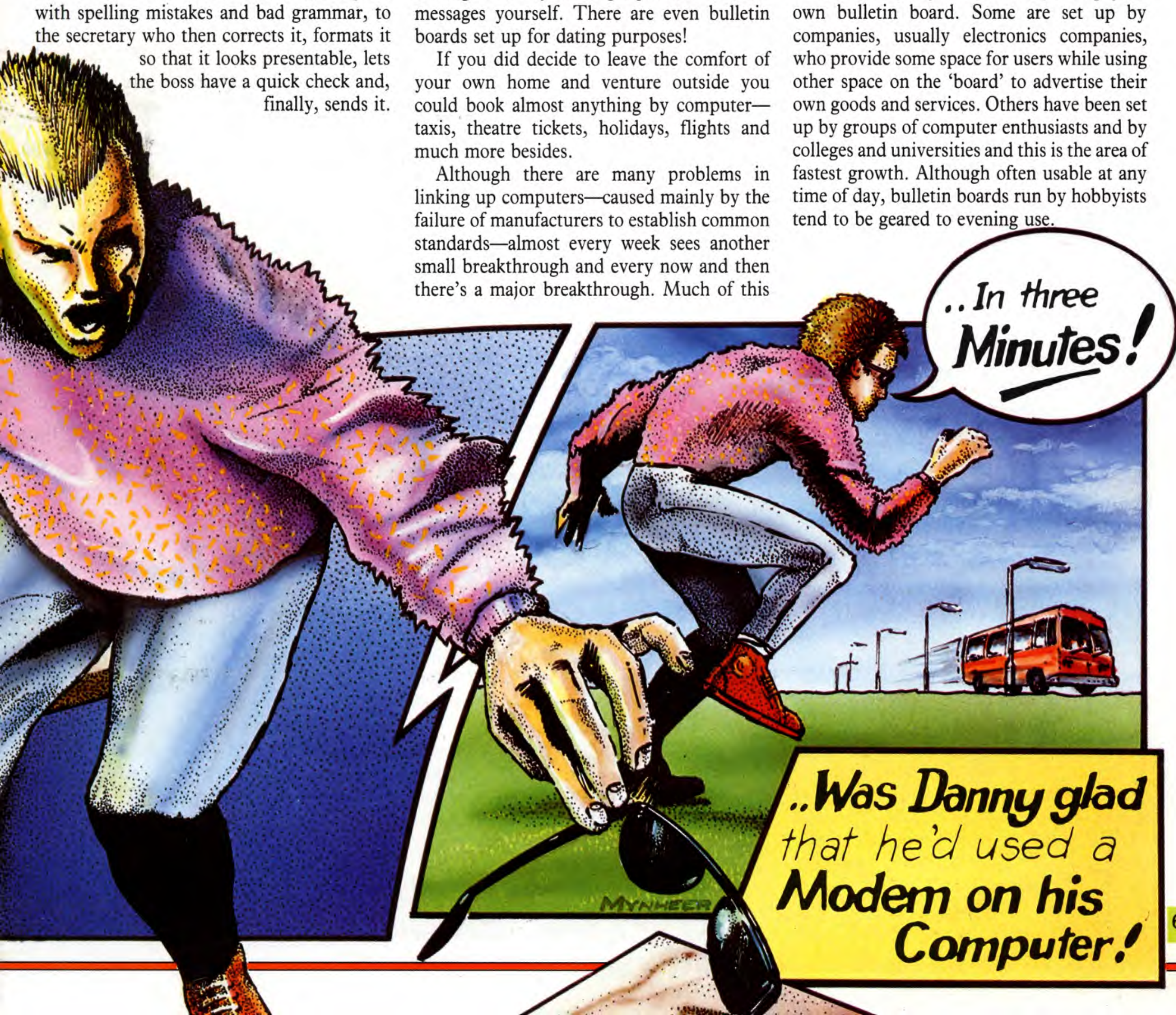
A lonely life? Perhaps, but not necessarily. It is also possible to contact other people with similar interests by computer. In some instances it doesn't matter where in the world they are, the cost of contacting them could be as little as a local telephone call. You can call up what is known as a *bulletin board* to read messages left by other people and to leave messages yourself. There are even bulletin boards set up for dating purposes!

If you did decide to leave the comfort of your own home and venture outside you could book almost anything by computer—taxis, theatre tickets, holidays, flights and much more besides.

Although there are many problems in linking up computers—caused mainly by the failure of manufacturers to establish common standards—almost every week sees another small breakthrough and every now and then there's a major breakthrough. Much of this pioneering work is carried out at the grass-roots level by enthusiastic amateurs and students and their teachers at universities and other higher education establishments.

The growth of interest in computer communications, and indications that it is the next exciting area for exploration, are shown in the fact that more and more bulletin boards are being set up.

You could, if you wanted to, set up your own bulletin board. Some are set up by companies, usually electronics companies, who provide some space for users while using other space on the 'board' to advertise their own goods and services. Others have been set up by groups of computer enthusiasts and by colleges and universities and this is the area of fastest growth. Although often usable at any time of day, bulletin boards run by hobbyists tend to be geared to evening use.



ARMCHAIR SHOPPING

Many people take the view that these uses of computer communications are so far-fetched that they are likely to remain in the realms of science fiction for many years to come. But everything described so far is possible today. Some people are so familiar with armchair banking and shopping that they regard it as a part of everyday life.

You can use your computer to access information about goods and prices at a wide range of shops. Or you can buy goods and services and pay for them with little more than a few keystrokes on the computer. The 'work' is done by a large central computer. This holds all the information and carries out the electronic transfer of money from the customer's account to the shop's account.

TELETEXT AND VIDEOTEX

You can link up to any of the growing number of *teletext* and *videotex* services even with a small home micro as long as it and your television set have been adapted properly. In the case of the UK's two current teletext services, ORACLE (run by the independent television companies) and CEEFAX (run by the BBC) this means you have access to hundreds of pages of information including program listings and programs that can be loaded directly into your computer. Direct downloading in this instance is usually only possible with a BBC computer connected to a special teletext decoder.

Sending and receiving—or *uploading* and *downloading*—software is one of the areas of greatest potential in computer communications. Software is already being transmitted by both TV and radio. But while software available through teletext can be directly downloaded, the software available from local radio stations must usually be recorded on tape first and then loaded in the normal way.

The major drawback of teletext, which uses broadcast signals, is that communication is one way only. But a viewdata system such as Prestel, which uses the telephone lines, can be two way. This means that your own machine can talk to the Prestel computer. In fact, there are several Prestel computers, offering the same service and, to some extent, interlinked, dotted around the UK. As well as making it much safer, this duplication of computers also helps to keep the system cheaper for the user who is charged realistic rates for using the telephone. The closer the computer is to the subscriber the cheaper the call—and many are on a local charge basis.

In conjunction with Prestel there are services such as Micronet which, as its name suggests, is a network for micro owners enabling them to talk to each other and also providing a software service for subscribers. Micronet uses the Telecom network but is produced by an independent publisher.

Another national computer communications network run, like Prestel, by British Telecom is Telecom Gold. Prestel is intended to be an information service and many users have a specially made keyboard dedicated solely to accessing the Prestel service. Telecom Gold, however, is much more flexible, developed especially for micro owners and therefore allowing a much greater degree of two-way communication.

It can, for instance, send and receive information at a wider range of speeds which means that it's accessible to more makes of micro. Subscribers to the system are given access, via a password, to a Prime 750 minicomputer. Users have all sorts of facilities at their disposal including Infox, which allows business subscribers to write their own programs to develop and manage a database, do spreadsheet calculations and write reports.

ELECTRONIC MAIL

Within systems like Telecom Gold it is possible to set up your own 'mailbox'. Anyone who wants to get in touch can write a letter and leave it stored in the computer for you to read at leisure. Both sender and receiver must have computers, of course! *Electronic mail* of this type is a powerful business facility which is a step up from simple bulletin boards.

NETWORKS

Most of the well known uses of computer communications involve many small computers having access to one large computer, but all sorts of combinations are possible.

A network could involve mainframe computers, minicomputers, microcomputers and the whole range of peripherals. The *New York Times*, for instance, uses all three types of computer in one of the most sophisticated newspaper production systems in the world. In this sort of system, several people working in one office may each use their own terminal linked to a minicomputer while one terminal in the office is linked to a mainframe.

The day-to-day work—in the case of a newspaper editorial office, writing news stories or features—can be carried out using the minicomputers while the mainframe computer is used for the administration of the company and for databases.

Meanwhile, journalists working away from the office—at home, for instance—can use a microcomputer with a modem to gain access to databases to gather information for their story over the telephone lines. Using a computer and a modem they can send a complete story to the office in a matter of seconds.

Many correspondents at baseball games now sit at a keyboard and screen rather than a typewriter. At the press of a couple of buttons their summary of the game and the result can be fed into the office computer. Investigative reporters can carry out spot checks on information by calling up the database on the mainframe computer.

Because manufacturers of computers have failed to agree on common international standards, there are problems in linking up computers but, one by one, these are being overcome and computers are becoming more and more compatible with each other. This is especially so in the case of computer communications using telephone lines, because of the way in which the signals are transmitted.

SECURITY

The fact that many different computers can now communicate with each other over the telephone lines delights most micro owners—but horrifies many universities, large companies and government agencies! The problem for many of these institutions is that they, too, use the telephone lines for internal and external connections for their big mainframe and minicomputers. This makes them vulnerable.

All sorts of elaborate security surrounds these big computers but in some cases it's been proved that the most sophisticated security is not enough. Computers at all sorts of organisations have been broken into. There have even been stories that people have accessed American military computers.

This was the basis of the film *War Games* and has been the subject of much controversy, especially in the USA where there have been complaints that hundreds of thousands of dollars worth of damage has been done by computer enthusiasts interfering with the mainframe computers of large corporations.

It's the potential offered by computer communications that has also led to fears of massive computer frauds netting the electronic criminals millions of pounds. Instead of jemmies or guns, robberies are committed with computers and telephones. Sometimes one of the biggest problems that banks and other vulnerable institutions have is finding out if they have been robbed at all! So much business is done by computers carrying out financial transactions electronically—using the telephone lines, of course—that detection of electronic crime is becoming more and more difficult. This is especially so if the criminal has managed to change all the records relating to his crime.

OTHER COMMUNICATIONS MEDIA

Telephones aren't the only means for communicating with other computers. Radio waves can also be used. The sort of short wave equipment used by radio hams can be used to open up a link between one computer and another—as long as both computers are connected up to equipment which is capable of transmitting and receiving radio waves.

The fastest communication between computers uses the technology of fibre optics. Information is coded in light pulses and sent down thin strands of 'glass'. This new technology is reckoned to be much more efficient than any present method, and it enables information to be carried at the speed of light. Fibre optic communications are already in use and there is no reason why the same principles should not be applied to computer communications as well.

Direct links between computers and other peripherals are most common in offices. More than two computers can be linked together to form a network which could, for instance, enable a number of computers to share the same facilities such as printers or disk drives.

TOWER OF BABEL

No matter what means of communication is chosen there is always the problem of compatibility. A micro can only communicate with another micro—or with any other peripheral device—through the right interface. Without the correct combination of cables and sockets and a code that both machines will understand, communication is impossible.

The telephone line is by far the most common medium for carrying messages between one computer and another over long distances. This can only accept a serial signal rather than the speedier parallel type.

This does not mean that all computers with parallel interfaces are unable to send signals via the telephone system. There is hardware and software available that will enable it to do so but it may be expensive—in some instances, more than the computer!

One of the problems with using a telephone for communication between computers is that computers carry information in discrete (separate) electrical pulses. But telephone systems are designed to carry the human voice—which is an analogue signal, continuous and variable. So data from a computer has to be converted into a similar signal.

MODEMS

The equipment used to convert the computer's data into signals which can be transmitted over the telephone lines, and vice versa, of course, is a modem. The word is an abbreviation of MOdulator/DEModulator, which is a description of the function of a modem.

There are two types of modem, the *acoustic coupler* and the *hard wired* or *direct connect* modem. Both come in a variety of shapes and sizes but are usually contained in a box. The acoustic coupler has two rubber cups which will accommodate the telephone handset.

The acoustic coupler is connected to the micro and turns signals from the micro into tones which can be sent down telephone lines. It also converts incoming tones from another computer into digital information which the receiving computer can understand.

Hard wired modems encode (or modulate) data from the computer directly into electrical signals and decode (or demodulate) incoming information into serial bits which can be understood by the computer. These hard wired modems can transmit information at faster speeds than acoustic couplers and are less prone to errors.

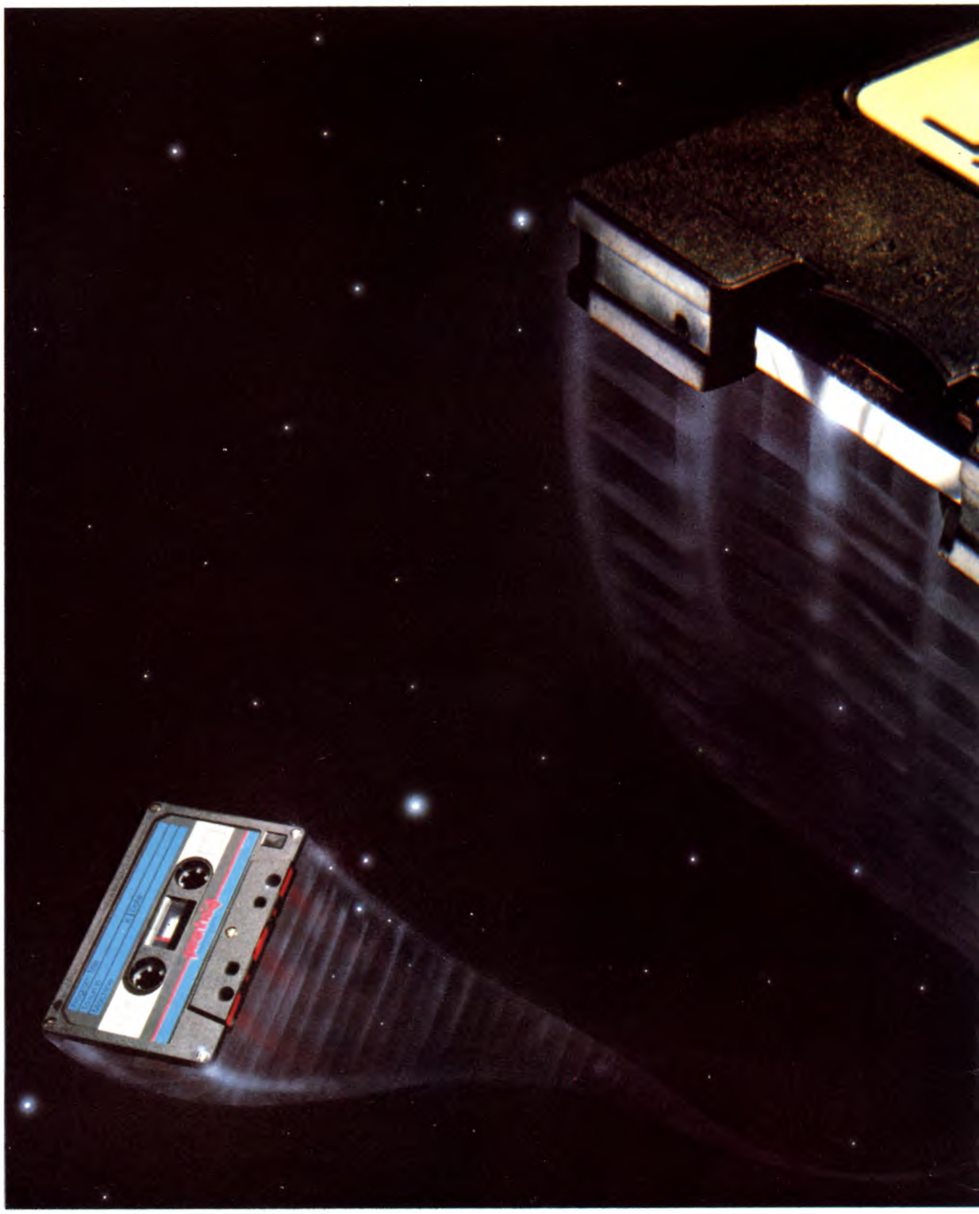
SPECTRUM MICRODRIVE CONVERTER

Spectrum programs that SAVE to and LOAD from tape can be made to work with a Microdrive by putting them through this simple machine code routine

So far, all the programs given in *INPUT* have assumed that you are SAVEing to and LOADing from tape. But more and more people are now using Microdrives or disk units.

Of course, if you do have a Microdrive you can modify the programs given in *INPUT* to work with them, by hand, yourself. But why not let your computer do it for you? The following assembly program will modify many BASIC, tape-dependent programs for use with the Microdrive on the 48K Spectrum. If you have a 16K machine, the program will still work on this. But you will need your own assembler since the *INPUT* assembler will not run on the 16K (unless of course, you want to hand assemble the program). And you will need to relocate the start address. The 16K origin should be at 32400 and you will have to CLEAR to 32399.

Similar programs can be written for other machines, but the BBC Micro does not need a conversion program as it will default to disk drives, if the equipment is present. The Electron and ZX81 don't have disk drives, and it is not possible to give a program for the Dragon as there are three different disk systems available. A disk program for the Commodore 64 and the Vic will follow.



To turn the normal tape instructions—LOAD, SAVE, VERIFY and MERGE—into the Spectrum Microdrive instructions ★"m",1; has to be added after the instruction, giving LOAD★"m",1; and SAVE★"m",1; and VERIFY★"m",1; and MERGE★"m",1;. And also, everything following the addition has to be shoved up in memory to make room. The following program makes the substitution and the shift for you. Following the explanation of how it works, you'll find detailed instructions on how to use it. Please refer to the Spectrum assembler modification on the inside back cover of Issue 18.

```
10 REM org 65200
20 REM ld hl,(23635)
30 REM start push hl
40 REM pop ix
50 REM inc hl
60 REM inc hl
70 REM inc hl
80 REM nxtchr inc hl
90 REM ld a,(hl)
100 REM cp 239
110 REM jr z,insert
120 REM cp 248
130 REM jr z,insert
140 REM cp 214
150 REM jr z,insert
160 REM cp 213
170 REM jr z,insert
180 REM cp 14
190 REM jr nz,notfp
200 REM ld de,5
210 REM add hl,de
220 REM notfp cp 13
230 REM jr nz,nxtchr
240 REM inc hl
```

- LOCATING **SAVES**, **LOADS**, **MERGES** AND **VERIFYS**
- AVOIDING FLOATING POINT NUMBERS
- USING THE STACK
- USING A DATA TABLE
- MOVING THE PROGRAM UP
- ADDING DATA
- UPDATING THE SYSTEM VARIABLES


```
250 REM push hl
260 REM ld de,(23627)
270 REM and a
280 REM sbc hl,de
290 REM pop hl
300 REM jr c,start
310 REM rst 8
320 REM defb 255
330 REM insert inc hl
340 REM ld a,(hl)
350 REM cp 42
360 REM jr z,nxtchr
370 REM push hl
380 REM ld hl,(23641)
390 REM pop de
400 REM push de
410 REM and a
420 REM sbc hl,de
430 REM ld b,h
440 REM ld c,l
450 REM ld hl,(23641)
460 REM ld de,13
470 REM add hl,de
480 REM ex de,hl
490 REM ld hl,(23641)
500 REM dec hl
510 REM dec de
520 REM lddr
530 REM ex de,hl
540 REM inc de
550 REM ld hl,table
560 REM ld bc,13
570 REM ldir
580 REM ld bc,13
590 REM pop hl
600 REM push hl
610 REM call $1664
620 REM ld e,(ix+2)
630 REM ld d,(ix+3)
640 REM ld hl,13
650 REM add hl,de
660 REM ex de,hl
670 REM ld (ix+2),e
680 REM ld (ix+3),d
690 REM pop hl
700 REM jr nxtchr
710 REM table defb 42
720 REM defb 34
730 REM defb 109
740 REM defb 34
750 REM defb 44
760 REM defb 49
770 REM defb 14
780 REM defb 0
790 REM defb 0
800 REM defb 1
810 REM defb 0
820 REM defb 0
830 REM defb 59
840 REM end
```

Q+A

What happens if I call this machine code program when there is no BASIC program in memory?

Don't worry. The machine code program will not crash. When there is no BASIC program there are no variables either. So the system variables PROG, VARS and ELINE all point to the same address. The next thing above ELINE memory is the edit line. In direct mode, the command is not transferred to the BASIC area when you press RETURN. So when you call the machine code program, the RANDOMIZE USR 65200 followed by the RETURN stays in edit line area.

The machine code routine then runs. It starts searching through the memory locations after PROG, which in this case is the edit line, for a LOAD, SAVE, MERGE or VERIFY. It won't find one. And when it hits the RETURN, it recognizes it as an end of line character. So it subtracts the value of VARS from the current byte pointer and goes onto the restart routine which returns to BASIC if there is no carry. But if the current byte pointer is past VARS, the subtraction will not give a carry, so the routine always returns to BASIC.

This program uses several of the important system variables which point to the BASIC area. These include PROG, VARS and ELINE which you should remember from the Spectrum memory map (see page 210). The pointer in 23,635 and 23,636, PROG, points to the first byte of the BASIC area. Normally this is fixed at 23,755, but when you plug in your Microdrive it shifts. So you have to look up the appropriate system variable.

The system variable, VARS, at 23,627 and 23,628, points to the first byte of the variables area—that is, the first byte past the end of the

BASIC program area in memory.

And ELINE, at 23,641 and 23,642, points to the first byte of the edit line, or the first byte past the end of the variables area.

HOW IT WORKS

The first assembly language instruction **ld hl,(23635)** loads the HL register pair with the address given by the system variable PROG. This is the address of the first byte of the BASIC program. This is then **push**ed onto the stack and **pop**ped off again into the IX register.

The HL register is going to be used as a pointer, travelling byte by byte through the program, while the IX register is going to be used to mark the beginning of each line of BASIC. This will be needed later when the byte carrying the line length has to be altered, which it will have to be when the additional instructions for accessing the Microdrive have been added.

This strange manoeuvre, **push**ing the pointer onto the stack and **pop**ping it off again, is used because there are no instructions that use both the HL and IX registers. This is because the HL and IX registers are equivalent. Instructions using the HL register use the IX register if a one byte prefix is specified before the opcode.

Each line of BASIC begins with the BASIC line number—which takes up two bytes—and the line length—which takes up another two bytes. Obviously, there are not going to be any tape instructions in this part of the line, so they can be skipped over by incrementing the HL register four times with the **inc hl** instruction.

The **ld a,(hl)** instruction then loads the accumulator with the first byte of the line proper. The next set of instructions compare it with the tokens for the various tape instructions which have to be changed.

THE CHECKS

The **cp** 239 compares the first byte with the token for a LOAD. And if it is a LOAD, the **jr z**—jump **r**elative if **z**ero—jumps to the **insert** routine.

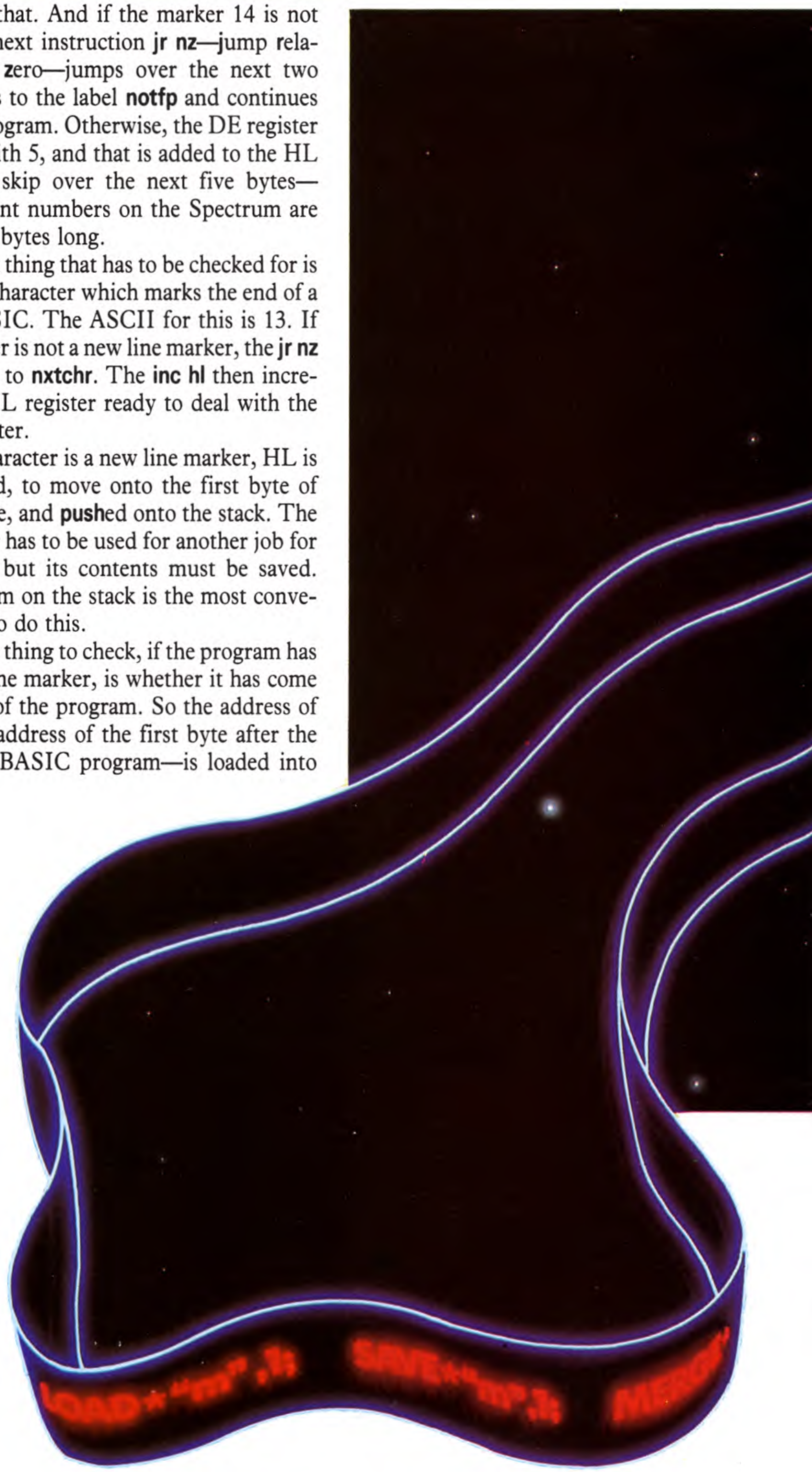
The **cp** 248 compares it with the token for SAVE. The **cp** 214 compares it with the token for VERIFY. And **cp** 213 compares it with the token for MERGE. If any of these occur, the processor jumps to the **insert** routine.

There is one other circumstance when these numbers could occur when they would not be the token for a tape instruction—in a floating point number (numbers stored in a scientific format, a topic to be covered later in *INPUT*). But all floating point numbers are always prefixed with the marker 14. So **cp** 14 checks for that. And if the marker 14 is not found the next instruction **jr nz**—jump **r**elative if **n**ot **z**ero—jumps over the next two instructions to the label **notfp** and continues with the program. Otherwise, the DE register is loaded with 5, and that is added to the HL register to skip over the next five bytes—floating point numbers on the Spectrum are always five bytes long.

The next thing that has to be checked for is a new line character which marks the end of a line of BASIC. The ASCII for this is 13. If the character is not a new line marker, the **jr nz** jumps back to **nxtchr**. The **inc hl** then increment the HL register ready to deal with the next character.

If the character is a new line marker, HL is incremented, to move onto the first byte of the next line, and **push**ed onto the stack. The HL register has to be used for another job for a moment, but its contents must be saved. Storing them on the stack is the most convenient way to do this.

The next thing to check, if the program has hit a new line marker, is whether it has come to the end of the program. So the address of VARS—the address of the first byte after the end of the BASIC program—is loaded into

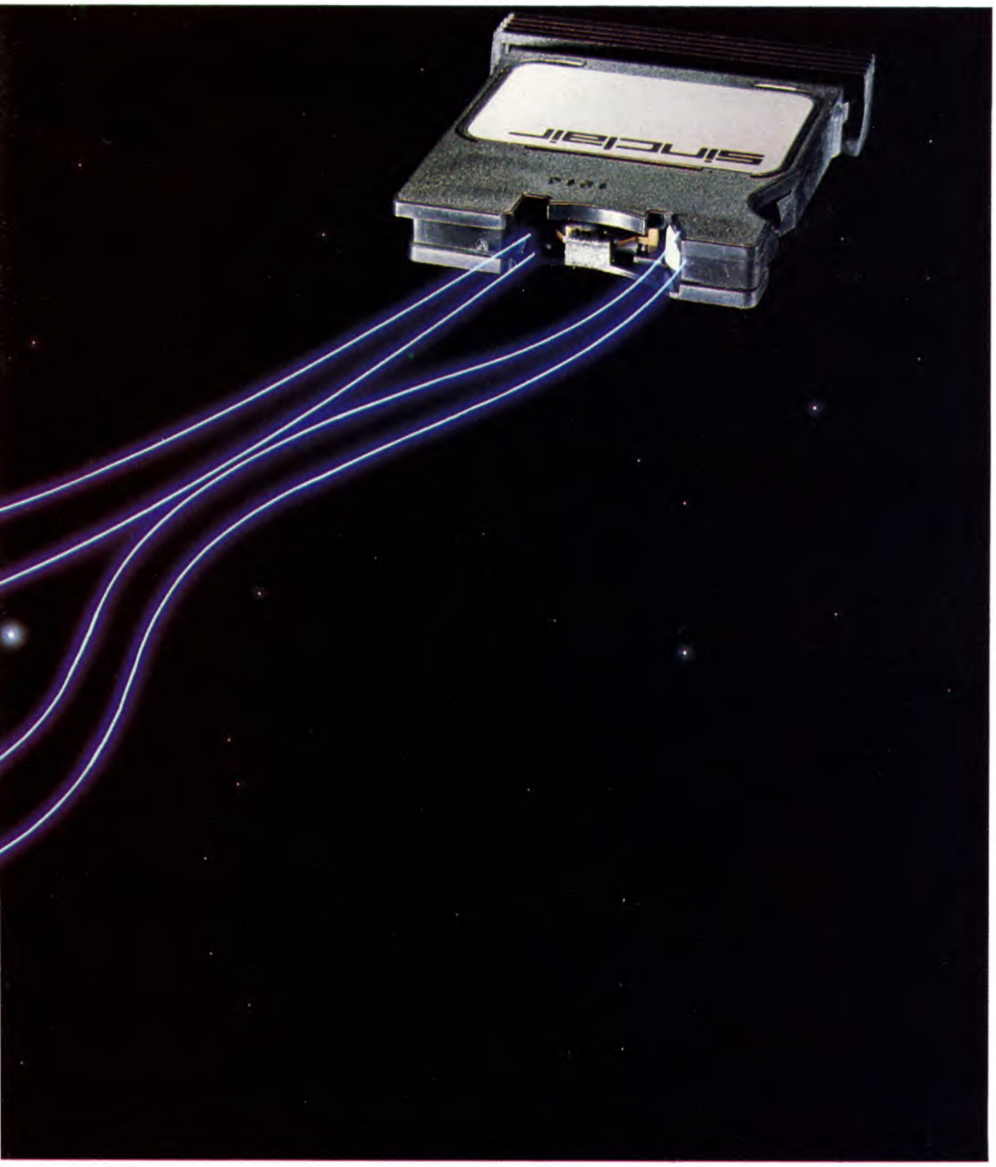


the DE register. The **and a** ands the accumulator with itself, which is a quick way of clearing the carry flag.

Then DE—which contains the value of VARS—is subtracted from HL—which carries the program pointer to the first byte past the last line of BASIC dealt with.

THE END OF BASIC

When the Microdrive conversion program has not reached the end of the BASIC program, the value of the pointer is bound to be less than that of VARS—so the subtraction will need a borrow and set the carry flag.



But when the end of the program has been reached, the pointer in HL will carry the same value as **VARS**, the subtraction will not require a borrow and the carry flag will not be set.

Before the result is tested, the original value of the pointer in HL is **pop**ped back off the stack—otherwise HL would carry the value after the subtraction.

The **jr c**—jump **r**elative if **c**arry flag set—then does the test. If the carry flag is set, and the end of the BASIC program has not been reached, the processor jumps back to the label **start**. The contents of HL are copied into IX and the routine starts on the new line.

If the carry flag is not set, and the end of the program has been reached, the jump is not made and the next two instructions are performed.

The **rst** 8 and **defb** 255 are an alternative way of returning to BASIC. **rst** 8 returns the computer to BASIC via the error message routine. This throws up the error message specified by the next byte in memory. **defb** is the assembler directive which **def**ines—or sets aside—a **b**yte of memory for the data. In this case, the number 255 is put into that location. The **rst** or **rest**art routine automatically increments whatever is given in this byte and returns that error message. It also clears the machine stack, so you don't have to worry about **push**ing and **pop**ping, as you would with a normal **ret** instruction. When it increments 255, or FF in hex, it gets Ø. The error message associated with Ø is OK—but with the interface connected you end up with a Program Finished error message.

THE INSERT ROUTINE

The next instruction starts the **insert** routine. And the first thing to be done is to check that the conversion for Microdrive has not already been done. You don't want to add the ★"m",1; twice. So the routine checks that the next byte is not a ★.

To do this, **inc hl** increments the HL register so that it looks at the next byte. This is then loaded into the accumulator by the **ld a,(hl)** and compared to 42—the ASCII code for ★—by **cp 42**. If the next character is a ★, the **jrz,nxtchr** jumps back to **nxtchr** and shifts the following characters along as normal—in other words, it does not go on to insert the Microdrive instruction, as it assumes it is already there.

If the next character is not a ★, the jump is not made and the next instruction is executed. This **push**es the HL register onto the stack because the HL register is going to be used for other jobs again and the pointer HL must be saved for use later.

HL is then loaded with the address of **ELINE**—the first byte past the end of the variables area—by **ld a,(23641)**. The next item on the stack is then **pop**ped off into the DE register. The last thing to be **push**ed onto the stack was the contents of the HL, which were then the current position pointer.

Once this pointer has been copied into the DE register, it is **push**ed back onto the stack, because it will be needed again later. But this **pop** and **push** does leave the value of the pointer in the DE register as well as on the stack.

The carry flag is cleared again with the **and a**, and the value of the pointer is subtracted from the value of **ELINE**. This gives the number of bytes left in the program and variables area after the tape instruction has been dealt with. The variables as well as the program are going to be shifted up in memory.

The **sbc hl,bc** puts the result of the subtraction in the HL register. The result is then stored in the BC register. This has to be done a byte at a time with the two instructions **ld b,h** and **ld c,l** because there is no **ld bc,hl** instruction. Well, you can't have everything.

The pointer **ELINE** is then loaded into the HL register again and the DE register is loaded with 13. The contents of these two registers are added to give the address **ELINE** has to be moved to allow the Microdrive additions to be made—they take 13 bytes, remember. The result of the **add** is put in the HL register and the **ex de,hl ex**changes the contents of the HL and DE registers, effectively storing the result of the addition in the DE register.

HL is then loaded with **ELINE** again. The **dec hl** and **dec de** then decrements the HL and DE registers, so that HL now points to the last byte in the variables area—in other words, the last byte that has to be moved—and DE contains the address, 13 bytes on, to which it has to be moved.

MAKING THE SHIFT

The **lddr**—**l**oad, **d**ecrement and **r**epeat—actually makes the shift. It loads the contents of the memory location pointed to by HL, into the memory location pointed to by DE (the easy way to remember which way round the shift goes is to think of DE as standing for DEstination), decrements HL, DE and BC and repeats the operation if BC is not zero. It is not hard to see that this one instruction will shift the contents of the variables area and the program byte by byte 13 bytes up in memory. And it will keep on doing so until it works its way down the tape instruction.

The **ex de,hl** exchanges HL and DE again, so the address of the tape instruction is now put into DE. This is incremented by **inc de** to point to the next byte after the tape instruction. The **lddr** instruction does the decrementing and testing after the last byte has been moved, so the pointers have been decremented one too many times. This **inc** compensates for that.

The HL register is then loaded with the start address of the table of data given at the end of the assembly listing here. As you can see these are stored in the form of **defb.** These are **def**ined **b**ytes. The 13 bytes stored there are the ★"m",1;. You'll see that the first six bytes are the ASCII codes for ★"m", and 1. And the last byte is the ASCII for ;.

Bytes seven to twelve contain the number 1 in floating point format. You'll note that it is prefixed by the token 14, for which the program checked early on. The following five bytes contain the actual floating point format version of 1. (Floating point numbers will be dealt with more fully in a later chapter.)

BC is then loaded with the number 13. It is going to be used as a counter to count these 13 bytes as they are stored in the space left by shifting the rest of the BASIC program up.

The **ldir**—**l**oad, **i**ncrement and **r**epeat—instruction loads the contents of the memory location pointed to by HL into the memory location pointed to by DE, increments HL and DE, decrements BC and repeats the process if BC is not zero. In other words, it reads the 13 bytes from the data table into space created after the tape instruction. So, for example, it converts SAVE into SAVE★"m",1;, to turn the tape SAVE instruction into the Microdrive version.

TIDYING UP

Once the conversion has been done, the system variables have to be updated. This is done by **call**ing the so-called 'pointers' routine in ROM which begins at 1664 in hex. But first BC must hold the number of bytes the pointers are to be changed by and HL must contain the value of the current byte. It only updates the system variables that relate to addresses above this point in memory.

The **ld bc,**13 loads BC with 13, and **pop hl** and **push hl** put the address of the current byte into the HL register, then back on the stack so it can be used again.

The next bit of tidying up to be done is fixing the line length. The second and third byte of each line of BASIC store the length of that line. The start address of the line was stored in the IX register at the beginning of the assembly language routine, remember. So **ld e,(ix+2)** and **ld d,(ix+3)** load the contents of the second and third bytes of the current line into the E and D registers. As you know these two registers are usually used as a register pair in the order DE. So the higher byte, the third, goes into D and the lower, the second, goes into E.

HL is then loaded with 13, and HL and DE are added. This boosts the line length by 13, but the result is put in HL and it needs to be in DE. There is no **add de,hl** instruction. Additions and other similar arithmetic functions can only be performed on the accumulator or HL register pair.

The **ex de,hl** moves the result into DE. It needs to be in DE to perform the next two instructions—**ld (ix+2),e** and **ld (ix+3),d.** There are no similar instructions with the L and H registers. There are no instructions using the HL and the IX registers together.

The **ld (ix+2),e** and **ld (ix+3),d** loaded the new, increased line length into the second and third bytes of the line. The current byte

pointer is then **pop**ped off the stack into HL, so that it is ready to be incremented to point to the next byte of the program when **jr nxtchr** jumps back to the label **nxtchr** and the whole process starts all over again.

HOW TO USE IT

Don't forget to CLEAR to one less than the origin to protect this program from overwriting. You'll need to do that both before you assemble it, and before you LOAD the program in again if you SAVE the object code on Microdrive or tape.

You should SAVE the source code along with the assembler, using the normal tape or Microdrive SAVE instructions, before you try testing the program. That way, if it crashes, you don't have to key the whole thing in again. All you have to do is LOAD the program in again and modify the assembly language.

Naturally, to test it you'll have to input a program that has tape instructions on it. The best one to use is the machine code monitor (see page 280). That way, if the program is working, you can use it to SAVE the machine code direct to Microdrive.

Otherwise, you can SAVE this program—and any other machine code program—to tape using:

SAVE "name" CODE start address, number of bytes

The "name" here is the name of the program and must be in quotes. start address is the origin—65,2ØØ was used here. And number of bytes is the length of the machine code. This program is 136 bytes long. The start address and the number of bytes must be separated by a comma.

To SAVE on Microdrive use:

SAVE★"m",1; "name" CODE start address, number of bytes

And to LOAD it back off tape you use:

LOAD "name" CODE

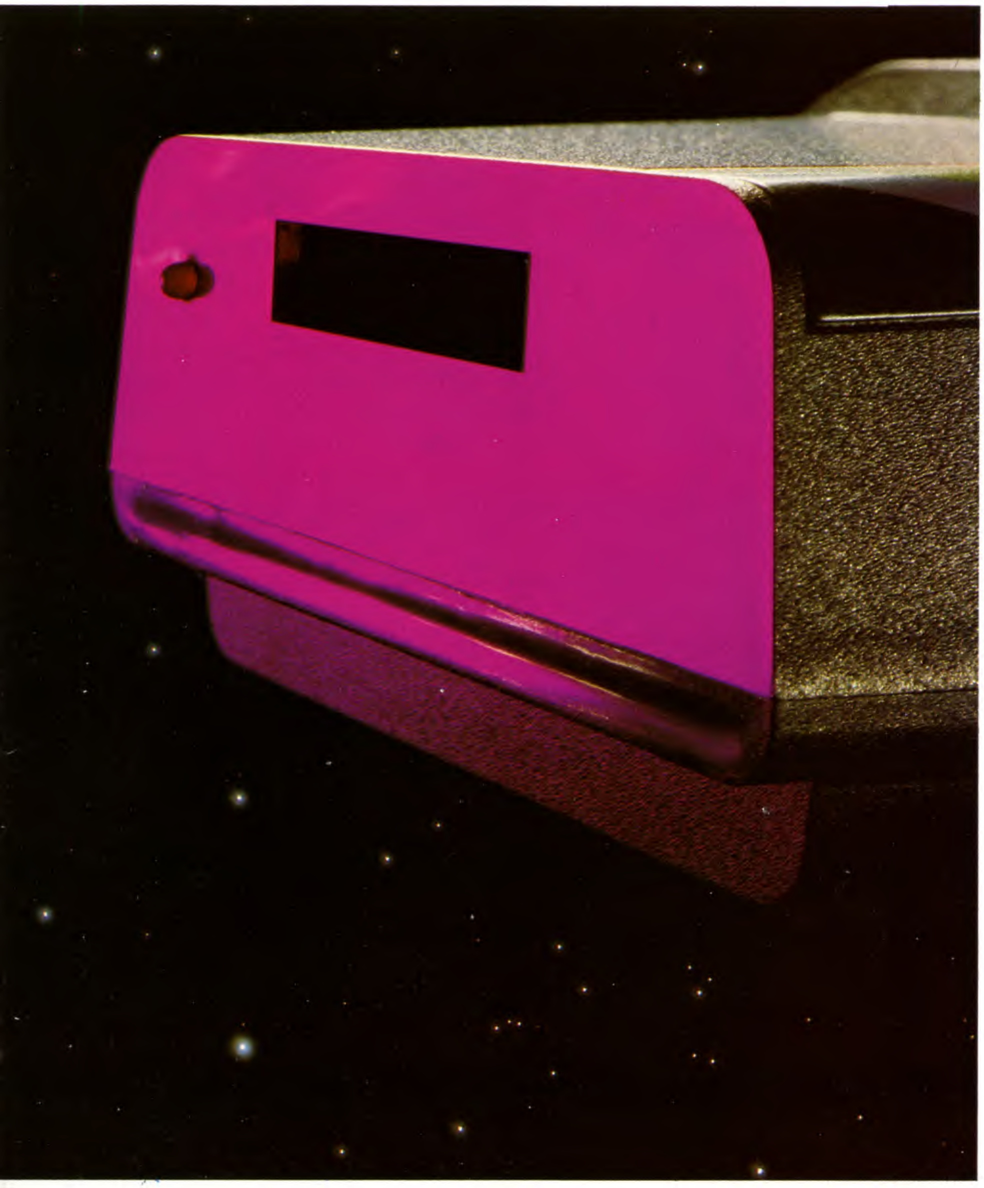
and to LOAD back into the machine from the Microdrive:

LOAD★"m",1; "name" CODE

You can put start addresses after the CODE in the LOAD instructions. If this is different from the place at which the program was assembled, the LOAD will relocate it, starting at the new address. This program will not relocate after it has been assembled though, because of the use of the label table which the assembler fills in with an absolute address.

To run the program, you use one of the regular machine code calls like:

RANDOMIZE USR 652ØØ



WARNING

You do have to be a bit careful when using programs originally designed for use with tape that have been converted for use with Microdrive. One problem is that on tape you can SAVE more than one program with the same name. In fact, when you're developing a program it is quite common—although bad practice—to SAVE each stage of development under the same file name. On Microdrive, though, you will get an error message if you try to do this without erasing the first file.

To do this you use the command:

ERASE"m",1;"filename"

where filename is the name of the file or program.

You also have to watch for null strings. A tape program might have LOAD A$ in it, for example. If A$ turns out to be a null string, with tape, LOAD "" will simply LOAD the next thing on the tape. LOAD★"m",1;"" will not work with a Microdrive though.

Your tape-dependent programs might also have PRINT instructions to tell you to position the tape and to press play, or play and record, on the tape recorder. You'll have to check through the program and remove these instructions by hand.

CREATING AND USING FILES

The word 'files' means many different things in computing, embracing everything from a loose descriptive term to specific ways of storing and accessing information

A good knowledge of what files are and how they are used can enormously extend the versatility of your computer.

File is the word used to describe any kind of data unit stored in a form which makes it accessible to your computer. Although an obvious application of the term is to the storage of information, even taping a BASIC program is, in this sense, creating a file. Filing can thus be applied to a BASIC program, a section of machine code, the 'text' produced by a wordprocessing program, or the raw data of filing programs.

FILE TYPES

File *types* take several forms. Each has different requirements and uses.

The easiest—and commonest—form of storing information is what's called a *serial* file. A file of this type consists of data that is read, item by item, from a storage device in exactly the same sequence as it was written.

Described simply, a serial file consists of three types of data: a *header* which among other things identifies the file; then there's a 'raw' data which goes to make up the file; and finally, (except on the Acorns) some kind of marker which identifies the end of the file.

One of the disadvantages of a serial file is that information can only be processed in the order in which it has been stored. This means that if your program has to look for something in the middle of a file you have saved, the whole lot has to be loaded in so the search can start at the beginning.

The situation is improved greatly if the information is actually arranged into some sort of order. In any database application, files are typically ordered alphanumerically. In this form, a serial file can correctly be called a *sequential* file, although this term is often used to describe *any* file that's been stored serially. But it's worth noting that a true sequential file has some sort of structure that is defined by the user or the program from which it came.

DISK-BASED FILES

The very great restriction of tape-based storage systems is that they can *only* read information serially, thus many tape-based files have to take the form of sequential files. For most common applications this is, thankfully, not a problem.

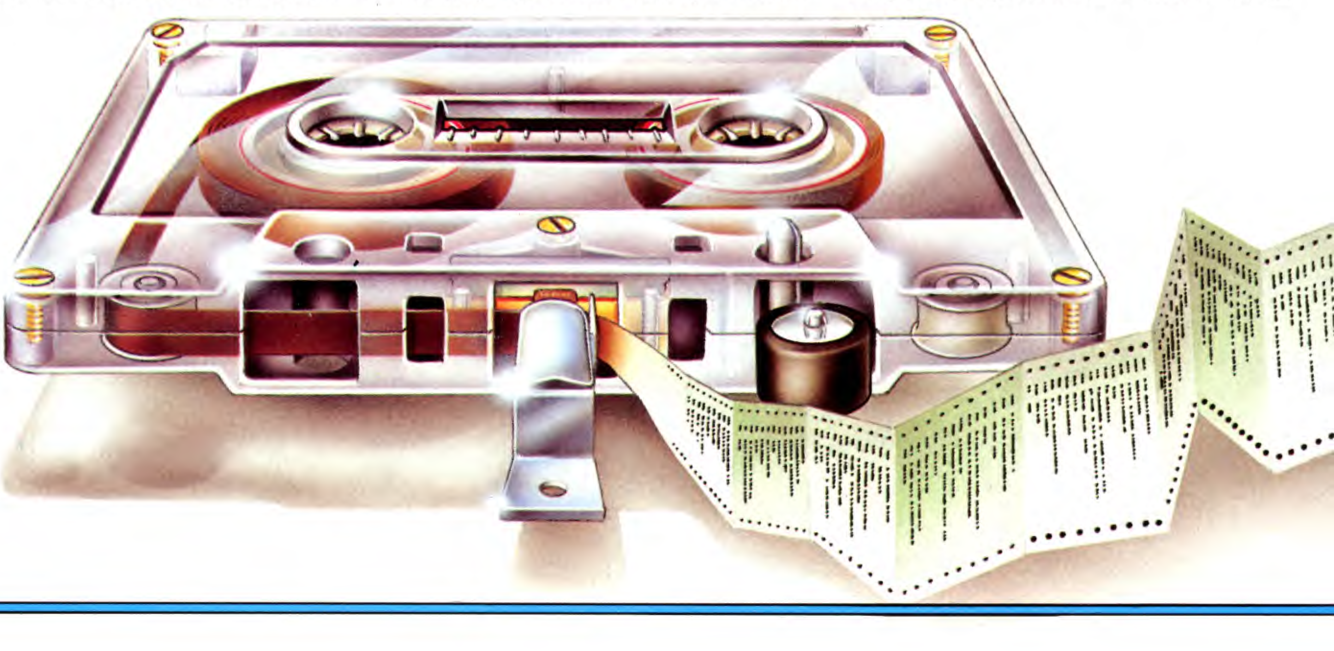
In fact, serial files are often used not just on tape, but also on the much more versatile disk-based storage systems (see page 504 to 508). However, the manner in which disk systems read data makes them suitable for, in particular, one other type of file: a *random access* file (otherwise called a *direct access* or *relative* file). There are other types of disk-based filing system, but these either are not used by home computers, or are peculiar to certain disk-operating systems.

A disk is formatted before use to give a certain number of *blocks* in which data can be stored. Each of these is a bit like a cell in a honeycomb, linked but clearly separate. The blocks are identifiable by track and sector coordinates; this enables them to be accessed directly and, if necessary, in any order (randomly) by a suitable program. This enables individual blocks to be called up, amended and saved once again without any of the others having to be disturbed—nothing else need be held in memory. As a result, information processing takes place very much faster and the actual size of the file itself is restricted only by the capacity of the disk system in use.

ASCII FILES

If file information is independent of a program and can be stored separately, you can have an almost infinite number of files. With a wordprocessing program, for example, you can have separate files for standard letters, articles, and so on.

But one of the real benefits of using a system of separate files is that different types of program can access information from 'foreign' files—and not necessarily on the same computer. A spreadsheet can access files for



- PROGRAM AND DATAFILES
- RESTRICTIONS OF TAPE
- INFORMATION TRANSFERS BETWEEN PROGRAMS
- SO WHAT IS A DATAFILE?
- WHAT WRITE AND READ MEAN TO A COMPUTER
- OPENING AND CLOSING FILES
- COMMANDS AND PROCEDURES USED BY YOUR COMPUTER

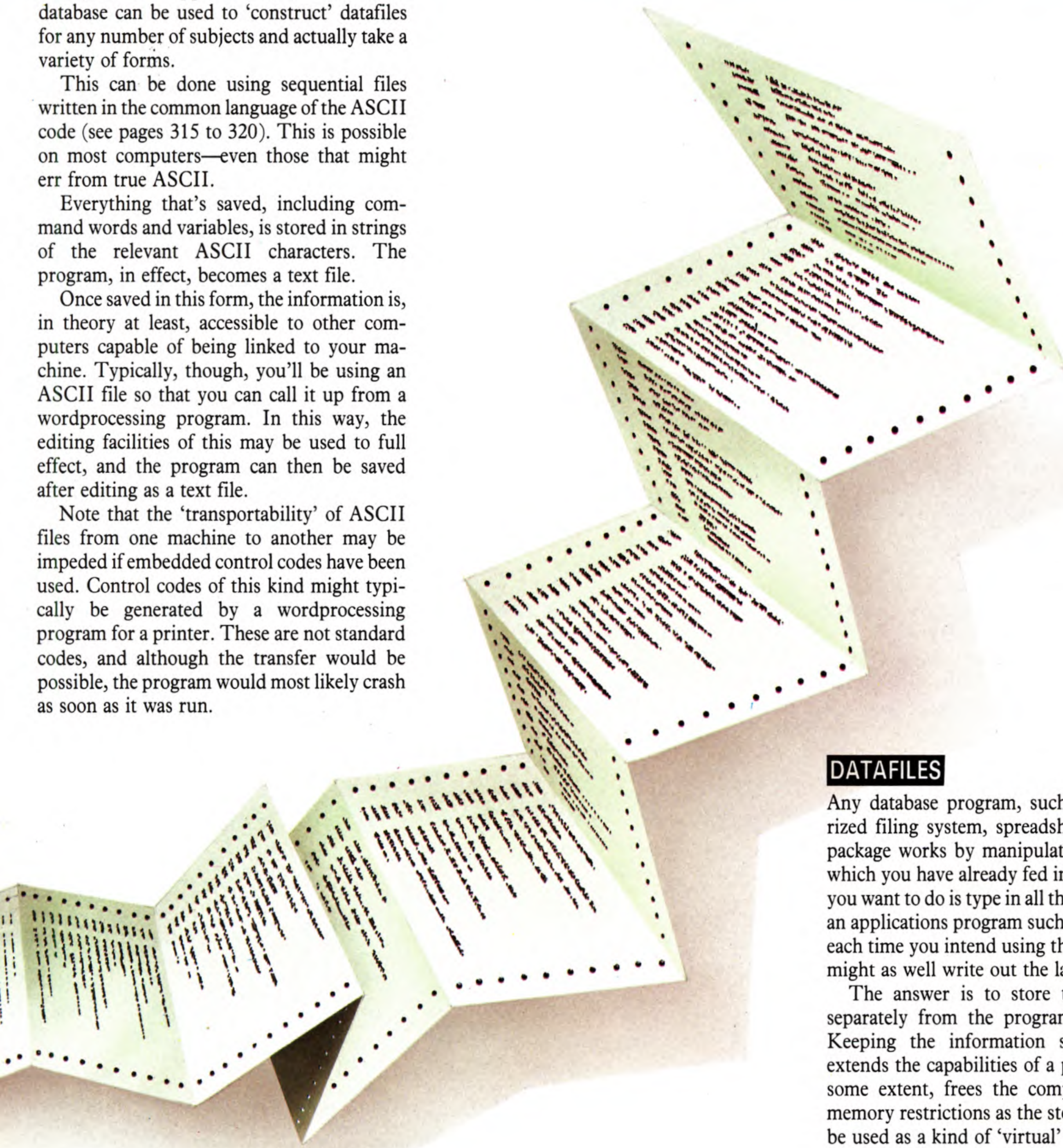
other, different applications, and a standard database can be used to 'construct' datafiles for any number of subjects and actually take a variety of forms.

This can be done using sequential files written in the common language of the ASCII code (see pages 315 to 320). This is possible on most computers—even those that might err from true ASCII.

Everything that's saved, including command words and variables, is stored in strings of the relevant ASCII characters. The program, in effect, becomes a text file.

Once saved in this form, the information is, in theory at least, accessible to other computers capable of being linked to your machine. Typically, though, you'll be using an ASCII file so that you can call it up from a wordprocessing program. In this way, the editing facilities of this may be used to full effect, and the program can then be saved after editing as a text file.

Note that the 'transportability' of ASCII files from one machine to another may be impeded if embedded control codes have been used. Control codes of this kind might typically be generated by a wordprocessing program for a printer. These are not standard codes, and although the transfer would be possible, the program would most likely crash as soon as it was run.



DATAFILES

Any database program, such as a computerized filing system, spreadsheet or accounts package works by manipulating information which you have already fed in. The last thing you want to do is type in all the information in an applications program such as a mailing list each time you intend using the program! You might as well write out the labels by hand.

The answer is to store the information separately from the program as a *datafile*. Keeping the information separate greatly extends the capabilities of a program and, to some extent, frees the computer from any memory restrictions as the storage device can be used as a kind of 'virtual' memory calling in only information as it is required.

The smallest component of any filing or

data system, computerized or not, is in fact a single *entry*. Several entries typically make up a *record*. Each of these records is divided into *fields*. Each field has a *label*, *title* or *heading* which remains more or less fixed for every record in that file—this simply indicates the nature of the entries made in the other part of the field. The record then forms part of, perhaps, a batch of similar records—a *file*. You can see this type of structure in operation in the file program on page 46.

Consider a simple card-index file. The file is like the complete set of cards, related by some overall subject—a club membership list perhaps. A record is an individual card, which could be for a single member of that club. And the fields are the separate entries on that card—the name, address, and so on.

In a 'paper' system, a file may be carried in a loose-leaf folder, box, cabinet—or whatever. All these 'containers' of records are often referred to as files but are essentially physical means of storing and (in some cases) transporting information.

It's not quite the same in a computer data system. Unless all the data is actually recorded on the same tape or disk it hasn't quite got the same 'physical' relationship as pieces of paper have to their filing cabinet. In some applications the data is completely stand-alone and would occupy a separate tape or disk. In others, the data must form an integral part of a program and cannot be considered a separate database.

Stand-alone datafiles (and in some cases integral datafiles) can often be called up by programs which were not originally responsible for their creation. Thus information entered via a spreadsheet program could be accessed by a wordprocessing program. Note the use of the word *accessed* rather than *transferred*—the program looks at and uses the information, and does not remove it. In a paper system, however, transference is the only way in which information can move from file to file without being copied.

FILE NAMES

What does not differ, however, is the need to describe a file by name. It would be difficult, perhaps impossible, to access data in a computerized system without a name for a program to hunt for.

Names for computer files have to be chosen with some care if a program is to access these properly. Tape data can be loaded on a 'next program in' basis if there is no name. But any search routine needs a name to latch on to.

On tape systems it is possible to have different files sharing the same name—or to save files with no name at all. But both pose severe problems. For example, if you instruct the computer to load the latest version of two files with the same name, it cannot make this decision, but will stop at the first it comes across. The solution is always to use a proper file name, and ensure that it is unique.

When you are using a disk-based system the *protocol* for the way you use file names is very much the same as for tape. But on some systems it is quite possible to overwrite a wanted file by unwittingly using the same name. Better disk operating systems prevent this happening. You can usually take special precautions to prevent this happening, such as *locking* the file.

In choosing a name make sure you stick to the length limits of each machine or system. On tape, a file name's length is restricted to 10 characters on the Spectrum, 16 on the Commodore, 10 for the Acorn and 8 for the Dragon/Tandy. On disk systems the length may differ.

Finally, use a file name which is easily remembered! It's a good idea to keep a separate record of what these are.

WRITING AND READING

The process of transferring data files to and from storage is described by the words *write* and *read*. Each computer has a different set of input/output commands for writing and reading but the general routine is the same for all.

In a typical application, a core program of some description is the first thing to be loaded into the computer—this might be a database or a wordprocessing program, for instance. Or, you could enter a program by hand.

If the program needs working data which is not available from within the program, this has either got to be loaded—or read in—from a data file or entered manually. This data is then manipulated, edited, amended or otherwise worked on. Data cannot be erased or amended while it is actually on a tape or disk. With some types of file, all the information has to be loaded into the computer, even if only a very small part of it has to be changed. This is the case with the Spectrum and Microdrive. In other cases, you can work on parts of the data by calling just what's needed.

Finally, the information is written out to an external storage device such as a tape or disk file, or another device such as a printer.

Let's look at the write and read stages more closely—in this order for convenience. First, an instruction must be given to the computer to *open a channel* so that it knows information is about to be passed somewhere. Other information can be provided along with the channel open command, to specify just what peripheral device is being addressed. Usually the peripheral will be a tape or disk unit.

After the channel has been opened, another *command* is used to change the output device if this is necessary. The computer has to know which device it is addressing regardless of whether information is coming in or going out. This other information is provided by additions to the main commands or use of alternative command words. On some computers, you must be careful if more than one device is fitted to your computer—as your computer may default to a particular value.

Commands are then used to *send* or write information to the nominated device. When you are reading information, the steps are similar up to this point but the commands for *receiving* or reading information obviously

reverse the direction of transmission. In each case, a buffer within the computer is used as a temporary information store, passing parcels of information onwards, piecemeal, normally only when the buffer is full.

When communications are complete, the opened channel is given the instruction to *close*. All information in the pipeline (within, in other words, a partially-filled buffer) is then forced to complete its journey, the computer marks the end of the file and generally tidies up.

No other information can be passed to or from a closed file until it is reopened. But this need not impede file handling on other opened channels in systems where several opened channels can operate at once.

While on the subject of terminology, several other words are used to describe the information transfers which take place when files are being written and read. For instance, external devices which receive data during a writing sequence are said to *listen*. Those that send data to the computer are said to *talk*.

The actual instructions used by the various computers do differ slightly, so now let's look at these. You will have come across most of them in programs you've already keyed in. Note that the normal SAVE and LOAD commands (and variants) which apply to *program* files are not listed here.

In each of the examples here the letter N stands for the file number which relates all the various input/output commands, and X or X$ is the data which passes along the file channels.

If you're using a cassette storage system, you are limited to using program files, bytes and arrays only. Therefore, the need to consider file handling procedures is, at this level, mainly of academic interest because the capability is so limited. True data files can, however, be handled rather more rapidly with the Microdrive fitted. The range of input/output keywords which can be used are:

OPEN#
This is used to prepare the system for transfer of information. It takes the standard form OPEN#N;"m";1;"filename" where the filename can be a string or previously assigned variable. Data is sent along a *stream* number N, along the Microdrive channel (m), to the Microdrive file.

PRINT#
Used in the form PRINT#N;X (or X$) to write data to the buffer.

INKEY$
This gets a single character or empty string from the keyboard each time it is used.

INPUT
Gets data from the keyboard until ENTER is pressed.

INPUT#
This is used in the form INPUT#N;X (or X$) to get data from the buffer until a carriage return is reached.

CLOSE#
This closes a previously opened stream using the instruction format CLOSE#N.

The instruction for opening up, using and then closing a channel on the Commodores can make use of the following BASIC 2 input (read)/output (write) keywords:

OPEN
Sets up a channel for input or output. It's used in the form OPEN N,D,S,X$ where the file number (in which N can range from 1 to 127) is followed by *device number* D. Each peripheral—and this includes modems and printers in addition to disk and tape storage units—has its own number (disk for example is usually 8, tape is 1, and the screen is 3).

This number is followed by a *secondary address* S, and then a string (X$). All but the file number can be optional. The secondary address is given a specific range of values depending on the device number used and a precise number within that range depending on the function of the OPEN statement.

The closing string can be a file name alone in the case of tape read/write operations, but for input/output to disk consists of drive number, file name, file type, and a specified read/write instruction. In the latter form, a typical reading instruction might take the form OPEN3,8,3,"Ø:FILENAME,S,R".

A great amount of control is possible from within an OPEN statement, as you can see. And, in reality, it takes a somewhat different form for each device.

CMD
Changes the output device number when this is necessary, typically to redirect information from the screen (to which the system defaults) to another device (whose number must be specified in a previous OPEN instruction).

GET
Gets one character at a time from the keyboard.

GET#
Gets one character as a time through an OPENed channel. It takes the form GET#N,X.

INPUT
Gets a data string from the keyboard.

INPUT#
Gets a data string through a previously OPENed channel, and takes the form INPUT#N,X. The string is assigned to the specified variable and assumed to be complete when a carriage return value is received.

PRINT
Directs information to the screen.

PRINT#
Directs information along a previously OPENed channel, to the screen, unless another device has been specified.

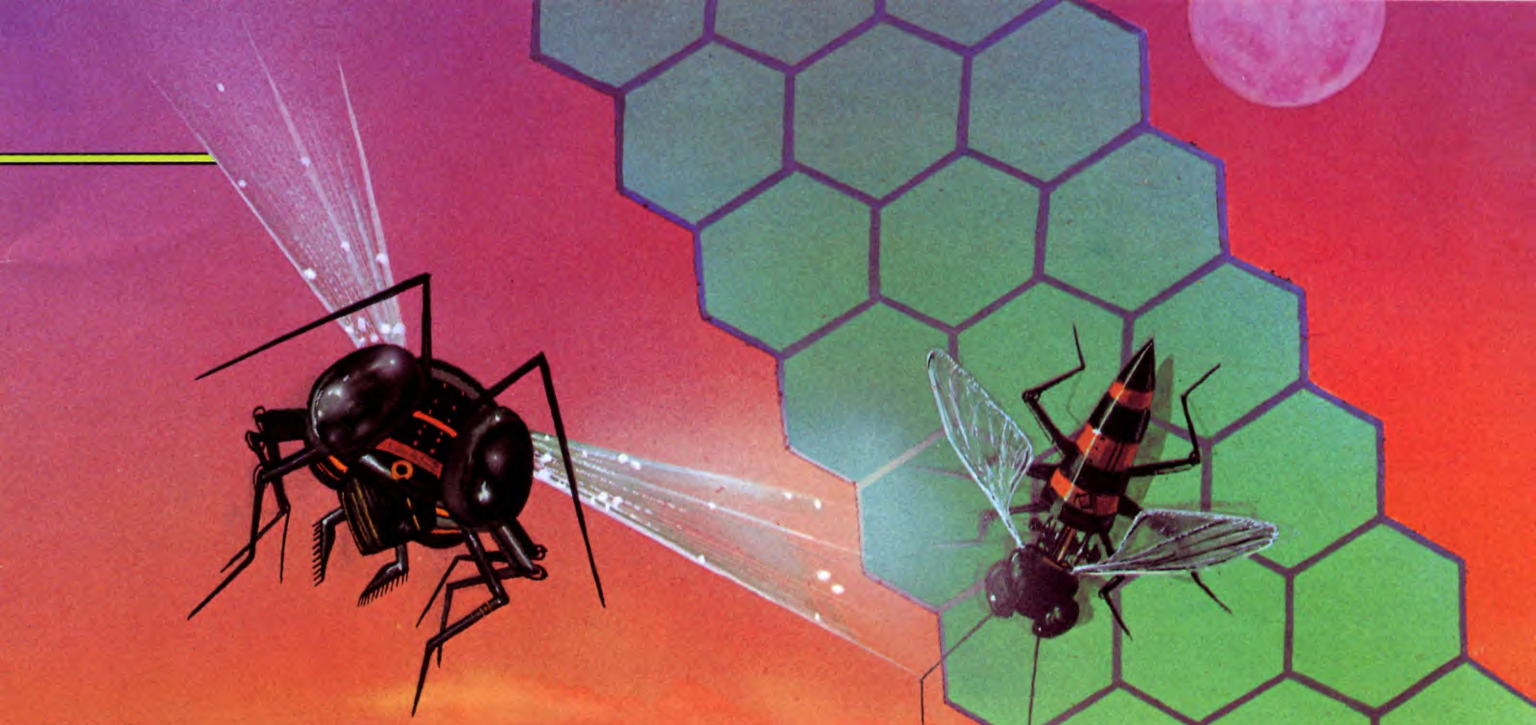
CLOSE
Closes a previously OPENed channel using the statement CLOSE N.

On the BBC micros a system of default filing is employed and any file-handling operation will be directed to or from the most 'senior' device fitted to the computer. Eligible devices generally will be tape and disk units but ROM, Econet and others may be fitted. Typically, though, all input and output will go to a disk rather than tape unit if both are fitted. The lesser device has to be specified when a channel is specially directed to it (for example, using ★TAPE or ★T. to pass information to the tape unit if a disk unit is fitted). On the Electron no such problem exists.

OPENIN
This sets up a channel for input (reading) or output (writing) in BASIC 1, but input only in BASIC 2. The instruction takes the form N=OPENIN("FILENAME") which opens a file called FILENAME and assigns its channel number to variable N. The file name can be a string or string variable and is optionally enclosed within brackets.

OPENOUT
Opens a new file to receive output, in both BASIC 1 and BASIC 2. It's used in the same way as OPENIN but for writing information.

OPENUP
Opens an *existing* file for input or output in BASIC 2 only. It follows the same form as OPENIN.



INPUT#

Gets a data string through a previously opened channel. It is used in the form INPUT#N,Z$ to read a string from a file with channel number N and assigns it to Z$.

PRINT#

Sends information out through a previously opened channel. The instruction takes the form PRINT#N,X where X is a number that is sent out.

CLOSE#

This closes a previously opened file. The operating system then marks the end of the file. The instruction takes the form CLOSE#N. Another form of this instruction is CLOSE#Ø which closes all open files.

Several straightforward keywords are used to open lines of communication with, typically, a tape recorder. Disk units can be fitted but the file-handling requirements of these are dictated by the operating systems in use.

OPEN

This opens a channel to a specified device. For writing (output) to tape it takes the form OPEN "O",#−1,"FILENAME" where only the FILENAME string is defined by the user's program. The device number (#−1) is that of a tape recorder. The input instruction from tape takes the basically similar form OPEN "I",#−1,"FILENAME".

INKEY$

Gets a single character from the keyboard.

INPUT#

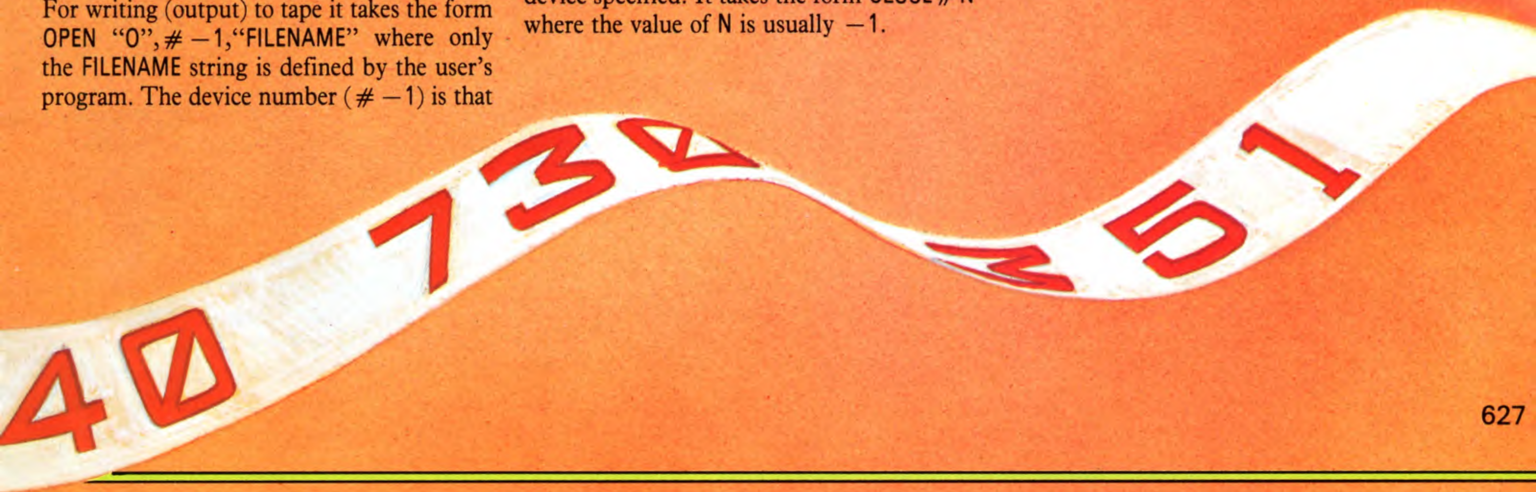
Gets string data from storage when a file is being read back. The additional instruction EOF(−1)—end of file—must precede the INPUT#−1 statement to avoid an input error. This simply makes sure that, when a device is talking, it doesn't continue to do so past the marker placed on the tape by the CLOSE#−1 instruction which terminated a prior writing operation.

PRINT#

This PRINTs information to device number N when the instruction takes the form PRINT#N,X$, where data for the string is obtained with an INPUT instruction. N is −1 for a tape recorder and −2 for a printer.

CLOSE#

This keyword closes communication to the device specified. It takes the form CLOSE#N where the value of N is usually −1.



TURN YOUR ADVENTURE INTO AN EPIC

If you're keen on adventure games, but frustrated by the limitations of your machine's memory, how about a program to fit the same amount of text into a lot less space?

The main problem with writing adventure games seems to be that there's never enough space for your latest masterpiece. Short of buying a new machine or a memory expansion there may seem little you can do except pull in the horizons of your adventure world, or simplify the program.

The only other way of easing the pressure on your straining memory locations is to try to find a way of making the program occupy less space. The usual methods for shortening programs, such as those on page 333, will not have too much effect, because most of the program is text. What you need, then, is some method of making text occupy less space.

Text is normally stored in computers as ASCII code—see pages 314 to 320. Using ASCII code will make each character occupy eight bits—one byte—of memory space.

If each character could be squeezed into less than eight bits, memory space would be saved. There are several possible routes open to you, each with advantages and drawbacks.

Probably the easiest way to compress text is to use only part of the ASCII coding. If you will be satisfied with a limited range of characters—upper case only, plus numbers and some punctuation—it's very easy to store each character in six bits.

Choosing a range of ASCII characters from 2Ø to 5F hex, for example, will give a fair range of characters for use in the game. If 2Ø hex is subtracted from each of the codes, the range will be reduced to Ø to 3F hex—which can be stored in six bits.

Using this kind of coding will enable you to reduce your memory requirements by a quarter—you'll be able to pack four characters into the space previously occupied by only three. In order to decode the stored characters, all that has to be done is to take each six bits in turn and add 2Ø hex to the number to turn it back to the original ASCII.

The main disadvantage with this kind of coding is that you are forced into choosing a range of 4Øh characters *before* you start programming, and you must stick to them—if you want a character outside the range, hard cheese! It could be that having the output in upper case only will be unacceptable to users of machines that normally output lower case to the screen—so you will need to make an alternative arrangement. Given these problems, a better alternative is needed.

One possibility approximates to how the Chinese language operates, assigning a unique character, or numeric value, to every word. This method of coding entails you deciding on the full vocabulary for your adventure, and assigning each word a number. Coding consists of comparing each of the words with a list of data, then storing each code in memory. Decoding consists of doing the reverse.

This 'Chinese Approach' offers very efficient use of memory, but needs to be rewritten for each new adventure. Of course, you can start off with a very large vocabulary and hope that you've chosen the right words for your needs, but that, in itself, is very wasteful of memory space. You may find yourself using some very large numbers as codes, and the list will take ages to scan.

The best alternative of all would be some kind of text compression system which will work with any adventure—in fact, on any text you wish to feed it. The compressor must make as efficient use as possible of the available memory, in terms of how it compresses the text and the amount of space the compression software itself occupies.

It's very important that the text compressor recognizes a full set of upper and lower case characters, numbers and punctuation.

Such a scheme requires a more radical approach from those outlined above—but it is possible to meet all the conditions, and a system of this type is the basis of the program which follows.

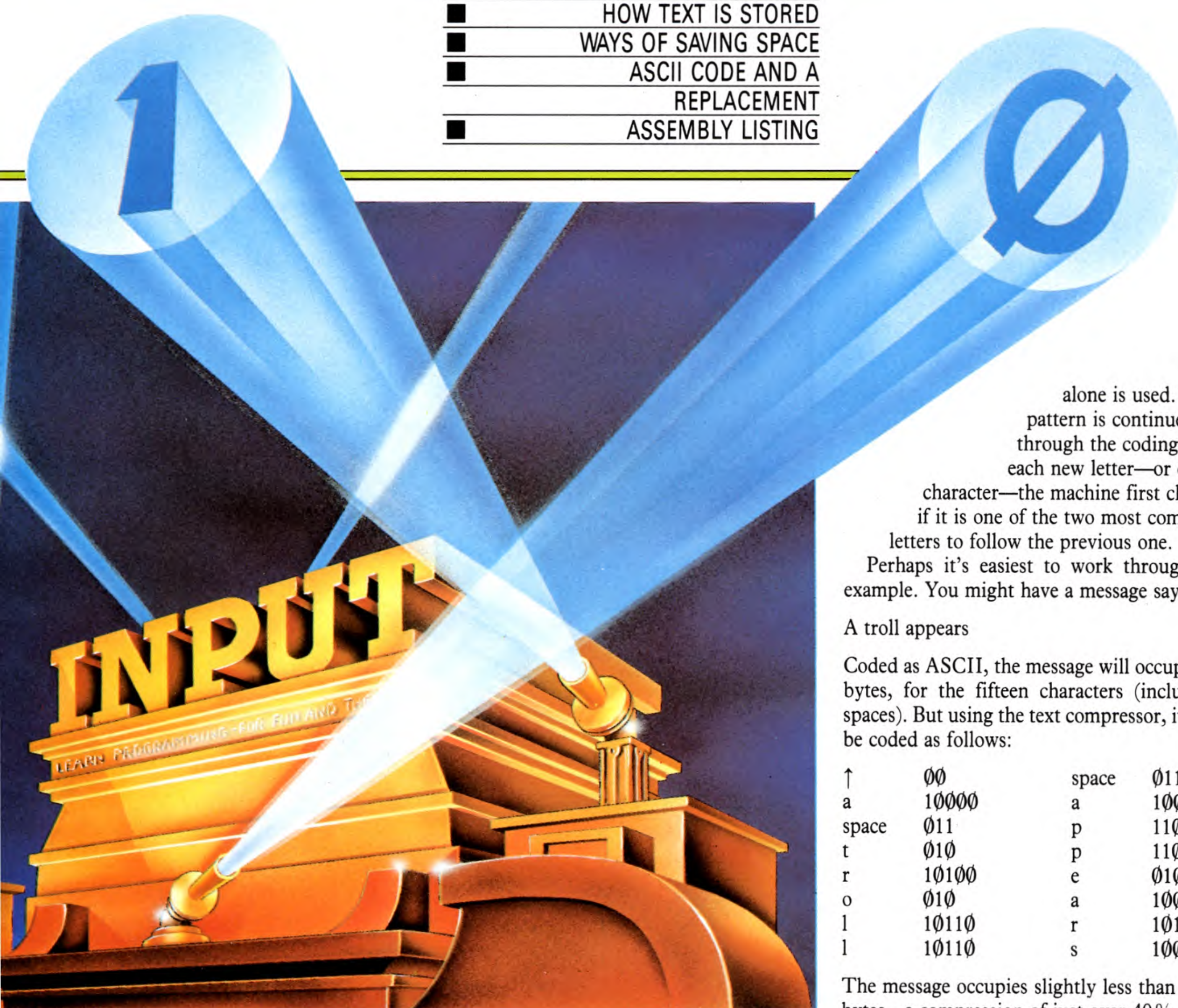
THE COMPRESSION SYSTEM

This text compressor borrows some information from the world of cryptography—the study of ciphers and codes. This tells us something about the structure of the text which is to be the subject of the compressor.

When code-breaking, it is very useful to know about how often certain letters occur in written English, as it enables the decoder to look for similar patterns in the coded words. To this end, cryptographers have laboriously plodded through large tracts of English text, counting how many times each letter, even pairs of letters, or whole words, occur.

The *INPUT* text compressor relies on the frequency of occurrence of single letters and, in certain cases, letter pairs. Binary numbers

- HOW TEXT IS STORED
- WAYS OF SAVING SPACE
- ASCII CODE AND A REPLACEMENT
- ASSEMBLY LISTING



are given to each of the letters of the alphabet—letters which occur often are given short, economical binary numbers, and letters which are least common are given the longest binary numbers.

One important by-product of using this kind of coding is that, if you test the text compression software with random, meaningless letters, there will quite possibly be little, or no, compression. If, on the other hand, you use English, you'll find that the compression is quite marked. Interestingly, the compression with, say, Spanish or German will be considerably less, because the patterns of occurrence will be quite different, requiring a different system of coding.

In addition to the codes for the letters occurring singly, the compressor also concerns itself with the two most common letters which may follow each letter. There is a summary of the codes in the table on page 636. You'll see that, for example, the most common letter following the letter T is H, and the second most common is I. There are special codes for these common following letters, again designed for economy.

When the compressor is encoding text, it remembers what the last letter was, and looks to see if the next letter is either the first or second most common following letter. If the letter *is* one of the common ones, the code at the top of the column is used. If the machine doesn't find one of the common letters, then the ordinary code for the letter occurring alone is used. This pattern is continued all through the coding. For each new letter—or other character—the machine first checks if it is one of the two most common letters to follow the previous one.

Perhaps it's easiest to work through an example. You might have a message saying:

A troll appears

Coded as ASCII, the message will occupy 15 bytes, for the fifteen characters (including spaces). But using the text compressor, it will be coded as follows:

↑	ØØ	space	Ø11
a	1ØØØØ	a	1ØØØØ
space	Ø11	p	11Ø11Ø
t	Ø1Ø	p	11Ø11Ø
r	1Ø1ØØ	e	Ø1Ø
o	Ø1Ø	a	1ØØØØ
l	1Ø11Ø	r	1Ø1ØØ
l	1Ø11Ø	s	1ØØØ1

The message occupies slightly less than nine bytes—a compression of just over 40%.

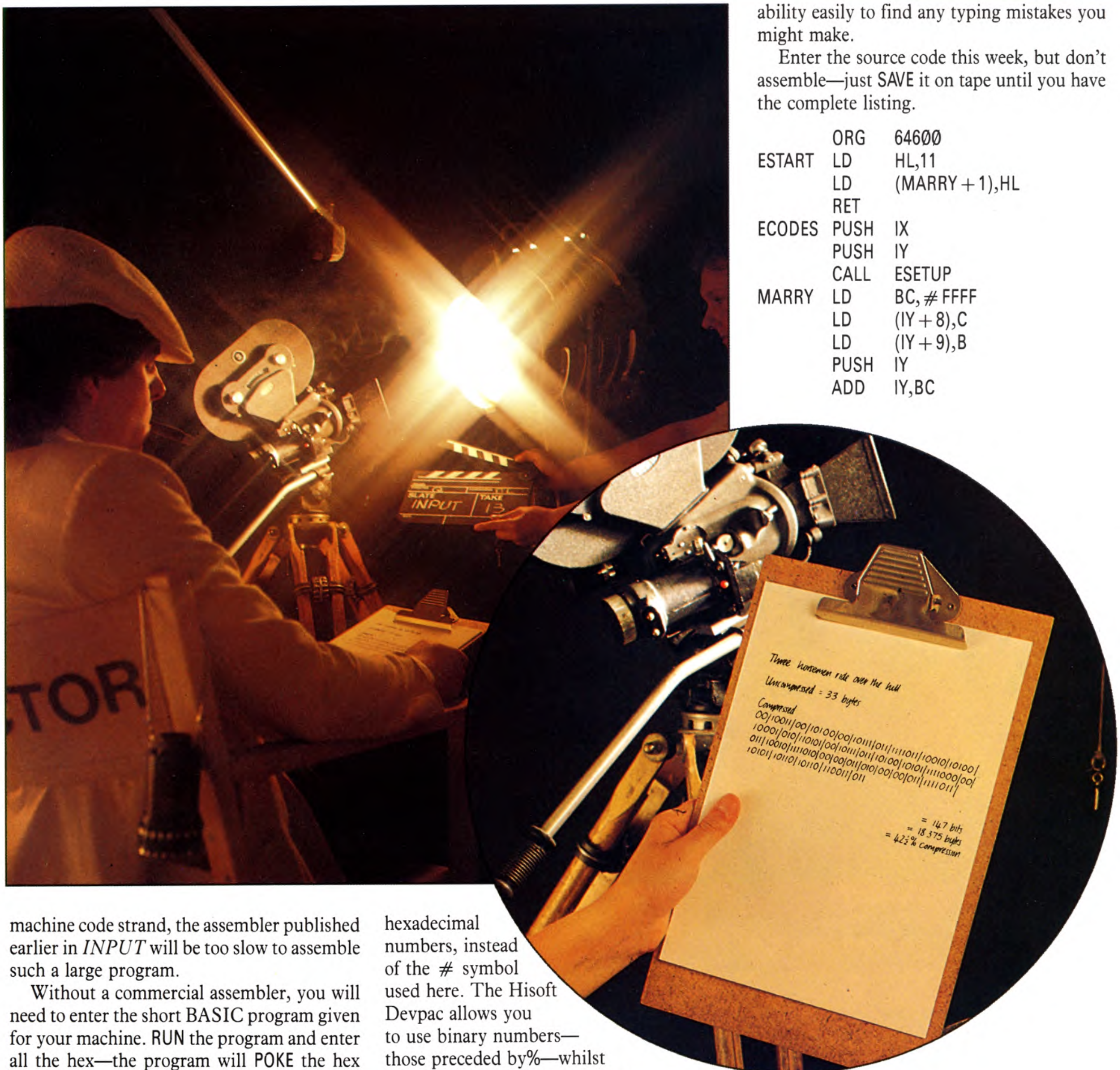
The codes for each of the available characters are in the table on page 636—notice that there are variations between the machines.

ENTERING THE COMPRESSOR

The text compressor has been written in machine code to speed up the coding and decoding of textual material.

The listing will be given as both hexadecimal and as assembly language. In this part of Games Programming, there is the assembly language listing for the coding part of the program. The decode part of the assembly listing will follow, along with the hexadecimal listing for the complete program.

You'll need a commercial assembler to use with the assembly listing or wait for the machine code listing. If you are following the



machine code strand, the assembler published earlier in *INPUT* will be too slow to assemble such a large program.

Without a commercial assembler, you will need to enter the short BASIC program given for your machine. RUN the program and enter all the hex—the program will POKE the hex into the correct memory locations. But more about that next time.

S

The text compressor was written using the Hisoft Devpac assembler, so you may find that your assembler won't recognise some of the following listing. To use another assembler, check that it will allow you to evaluate arithmetic expressions—if your assembler will not, it's probably easiest to enter the hex listing which follows later in *INPUT*.

Other variations between assemblers include using h following a figure to designate hexadecimal numbers, instead of the # symbol used here. The Hisoft Devpac allows you to use binary numbers—those preceded by%—whilst in others you'll have to evaluate the numbers yourself, because the assembler doesn't understand binary. EQU isn't supported by some assemblers, so try omitting that part entirely—just have a line containing the label and nothing else. Finally, some assemblers recognise a non-standard form of ADD—try replacing ADD A,8 with ADD 8, for example.

If you aren't sure how to convert the listing so that it's compatible with your assembler, or it seems a great deal of trouble to do the conversion, you'd probably be better off using the hexadecimal listing, sacrificing the ability easily to find any typing mistakes you might make.

Enter the source code this week, but don't assemble—just SAVE it on tape until you have the complete listing.

```
        ORG     64600
ESTART  LD      HL,11
        LD      (MARRY+1),HL
        RET
ECODES  PUSH    IX
        PUSH    IY
        CALL    ESETUP
MARRY   LD      BC,#FFFF
        LD      (IY+8),C
        LD      (IY+9),B
        PUSH    IY
        ADD     IY,BC
        LD      C,(HL)
        INC     HL
        LD      B,(HL)
        PUSH    HL
        INC     HL
        ADD     HL,BC
        LD      (ECTEST+1),HL
        LD      A,7
        LD      (BITCNT+1),A
        LD      HL,CSP
        LD      (PREFCD+2),HL
ECLOOP  POP     HL
        INC     HL
```

```
        PUSH    HL
        LD      A,(HL)
        AND     A
ECTEST  LD      BC,#FFFF
        SBC     HL,BC
        JR      Z,ECLEND
        CALL    ENCODE
        ADD     A,Ø
        JR      Z,ECLOOP
ECLEND  POP     HL
        LD      A,Ø
        CALL    ENCODE
        INC     IY
        PUSH    IY
        POP     HL
        POP     BC
        LD      A,(BITCNT+1)
        ADD     A,249
        SBC     HL,BC
        LD      (MARRY+1),HL
        POP     IY
        POP     IX
        RET
ENCODE  LD      B,CLAST-CODE+1
        CP      "@"
        JR      C,LAKE
        XOR     #2Ø
LAKE    LD      C,A
        LD      HL,CLAST
TRYC    SUB     (HL)
        JR      Z,MATCH
        CP      #2Ø
        JR      NZ,NOUPP
        LD      A,"↑"+#2Ø
REDO    PUSH    HL
        PUSH    BC
        CALL    ENCODE
        POP     BC
        POP     HL
        JR      MATCH
NOUPP   CP      #EØ
        JR      NZ,NOLOW
        LD      A,"__"+#2Ø
        JR      REDO
NOLOW   LD      A,C
        DEC     HL
        DJNZ    TRYC
        RET
MATCH   LD      A,B
PREFCD  LD      IX,#FFFF
BITCNT  LD      E,#FF
        DEC     E
        DEC     E
        DEC     A
        JR      Z,SPACEE
        CP      (IX+FIRST-CODE)
        JR      NZ,NOTFIR
        LD      A,Ø
        JR      TOP
NOTFIR  DEC     E
        CP      (IX+SECOND-CODE)
        JR      NZ,LONG
        LD      A,%Ø1Ø *32
        JR      TOP
SPACEE  LD      A,%Ø11 *32
        DEC     E
        JR      TOP
LONG    DEC     E
        DEC     E
        DEC     E
        DEC     E
        DEC     E
        ADD     A,255+CODE-CLAST
        CP      255+CEIGHT-CLAST
        JR      NC,TOP
        INC     E
        SLA     A
        ADD     A,CLAST-CEIGHT+1
        CP      255+CSEVEN-CLAST
                +CSEVEN-CEIGHT
        JR      NC,TOP
        INC     E
        SLA     A
        ADD     A,CLAST-CSEVEN
                +CEIGHT-CSEVEN+1
        CP      255+CSIX-CSEVEN
                +CSIX-CSEVEN+CSIX
                -CEIGHT+CSIX
                -CLAST
        JR      NC,TOP
        INC     E
        SLA     A
        ADD     A,CLAST-CSIX+
                CEIGHT-CSIX+
                CSEVEN-CSIX+
                CSEVEN-CSIX+1
TOP     LD      C,A
        LD      A,(HL)
        CP      "↑"
        JR      Z,TOPOUT
        LD      A,(IX)
        CP      "__"
        JR      NZ,HOLD
        LD      HL,CPUN
HOLD    LD      (PREFCD+2),HL
TOPOUT  LD      A,C
TOPACT  LD      BC,(BITCNT+1)
        LD      B,C
        INC     B
JAR     RLCA
        DJNZ    JAR
        LD      IX,LO
        ADD     IX,BC
        LD      B,A
        AND     (IX+1)
        OR      (IY)
        LD      (IY),A
        LD      A,B
        AND     (IX+UP-LO+1)
        LD      (IY+1),A
        LD      A,E
        CP      %1ØØØØØØØ
        JR      C,DONE
        ADD     A,8
        INC     IY
DONE    LD      (BITCNT+1),A
        LD      A,Ø
        RET
CODE    EQU     $
CSP     DEFB    "□"
CA      DEFB    "A"
CS      DEFB    "S"
CO      DEFB    "O"
CT      DEFB    "T"
CR      DEFB    "R"
CI      DEFB    "I"
CL      DEFB    "L"
CE      DEFB    "E"
        DEFB    "C"
CSIX    EQU     $
CUP     DEFB    "↑"
CAR     DEFB    "__"
CU      DEFB    "U"
        DEFB    "M"
Cp      DEFB    "P"
        DEFB    "W"
        DEFB    "Y"
CN      DEFB    "N"
        DEFB    "B"
        DEFB    "G"
CSEVEN  EQU     $
CD      DEFB    "D"
CF      DEFB    "F"
        DEFB    "V"
CH      DEFB    "H"
CEIGHT  EQU     $
        DEFB    "K"
        DEFB    "Q"
CX      DEFB    "X"
        DEFB    "J"
        DEFB    "Z"
        DEFB    "["
CPO     DEFB    "\"
CLAST   DEFB    "]"
CPUN    EQU     $
FIRST   DEFB    CUP-CODE
        DEFB    CN-CODE
        DEFB    CT-CODE
        DEFB    CN-CODE
        DEFB    CH-CODE
        DEFB    CE-CODE
        DEFB    CN-CODE
        DEFB    CE-CODE
        DEFB    CR-CODE
        DEFB    CH-CODE
        DEFB    CS-CODE
        DEFB    CL-CODE
        DEFB    CN-CODE
        DEFB    CE-CODE
        DEFB    CO-CODE
        DEFB    CA-CODE
        DEFB    CA-CODE
        DEFB    CD-CODE
```

```
        DEFB   CE-CODE
        DEFB   CE-CODE
        DEFB   CE-CODE
        DEFB   CT-CODE
        DEFB   CE-CODE
        DEFB   CE-CODE
        DEFB   CE-CODE
        DEFB   CU-CODE
        DEFB   Cp-CODE
        DEFB   CU-CODE
        DEFB   CE-CODE
        DEFB   CAR-CODE
        DEFB   CAR-CODE
        DEFB   CSP-CODE
        DEFB   CAR-CODE
SECOND  DEFB   CT-CODE
        DEFB   CT-CODE
        DEFB   CE-CODE
        DEFB   CF-CODE
        DEFB   CI-CODE
        DEFB   CO-CODE
        DEFB   CT-CODE
        DEFB   CI-CODE
        DEFB   CD-CODE
        DEFB   CO-CODE
        DEFB   CT-CODE
        DEFB   CN-CODE
        DEFB   CS-CODE
        DEFB   CA-CODE
        DEFB   CE-CODE
        DEFB   CH-CODE
        DEFB   CO-CODE
        DEFB   CT-CODE
        DEFB   CL-CODE
        DEFB   CH-CODE
        DEFB   CI-CODE
        DEFB   CO-CODE
        DEFB   CI-CODE
        DEFB   CA-CODE
        DEFB   CI-CODE
        DEFB   CX-CODE
        DEFB   CT-CODE
        DEFB   CO-CODE
        DEFB   CI-CODE
        DEFB   CUP-CODE
        DEFB   CPO-CODE
        DEFB   CAR-CODE
        DEFB   CUP-CODE
LO      DEFB   Ø
        DEFB   %1
        DEFB   %11
        DEFB   %111
        DEFB   %1111
        DEFB   %11111
        DEFB   %111111
        DEFB   %1111111
UP      DEFB   %11111111
        DEFB   %1111111Ø
        DEFB   %111111ØØ
        DEFB   %11111ØØØ
        DEFB   %1111ØØØØ
        DEFB   %111ØØØØØ
        DEFB   %11ØØØØØØ
        DEFB   %1ØØØØØØØ
        DEFB   Ø
ESETUP  LD     HL,(23627)
        LD     D,Ø
        LD     C,2
EREPEAT LD     A,(HL)
        CP     #EØ
        JR     NC,VFOR
        CP     #CØ
        JR     NC,VSA
        CP     #AØ
        JR     NC,VMN
        CP     #8Ø
        JR     NC,VNA
        CP     #6Ø
        JR     NC,VN
        JR     VS
VFOR    LD     E,19
        ADD    HL,DE
        JR     EREPEAT
VMN     INC    HL
        LD     A,(HL)
        CP     #EØ
        JR     C,VMN
VN      LD     E,6
        ADD    HL,DE
        JR     EREPEAT
VS      CP     "Z"
        JR     NZ,VSA
        PUSH   HL
        DEC    C
        JR     Z,FINDEX
VSA     INC    HL
        LD     E,(HL)
        INC    HL
        LD     D,(HL)
        INC    HL
        ADD    HL,DE
        LD     D,Ø
        JR     EREPEAT
VNA     CP     "z"+#2Ø
        JR     NZ,VSA
        PUSH   HL
        POP    IY
        DEC    C
        JR     Z,FINDEX
        JR     VSA
FINDEX  POP    HL
        INC    HL
        RET
```

The assembly listing that follows is for the Commodore 64, but if you own an expanded Vic 20 you can easily adapt it so that it will run in your machine. In the assembly listing, the parts to change have been printed in bold type, so look in the table for the equivalent.

Commodore 64	Vic 20 expansion +8K	+16K	+24K	+32K
CD	3D	5D	7D	BD
CE	3E	5E	7E	BE
CF	3F	5F	7F	BF

Vic 20 owners with 8, 16 or 24K expansions must also change two pointers. **POKE** the two memory locations first.

Vic 20 + 8K	POKE 52,61	POKE 56,61
Vic 20 + 16K	POKE 52,93	POKE 56,93
Vic 20 + 24K	POKE 52,125	POKE 52,125

The Commodore 64 program starts at **CD14**, so if you have a Vic 20 you'll have to change the **CD** to the equivalent for your machine. The assembly listing is wrongly ordered. It should start at **LDX #$1F**, 20 lines from the bottom of column 1, page 633. Omit the final **PHA**. Then enter the lines from the beginning down to **RTS**.

Don't assemble the program until you have the complete listing, but just **SAVE** the source code on tape—remember, the second half of the listing is still to come. You will need to tell the assembler the start address for the code.

Commodore 64	CD14
Vic 20 + 8K	3D14
Vic 20 + 16K	5D14
Vic 20 + 24K	7D14
Vic 20 + 32K	BD14

```
PHA
TYA
PHA
LDY  #$Ø8
STY  $CE1A
LDY  #$ØØ
STY  $CE13
LDA  ($2D),Y
INY
CMP  #$5A
BNE  $CDEØ
LDA  ($2D),Y
CMP  #$8Ø
BEQ  $CDE6
TYA
ADC  #$Ø6
TAY
BNE  $CDD3
INY
STY  $CE2F
PLA
TAY
PLA
RTS
PHA
TXA
PHA
TYA
PHA
LDA  $2F
STA  $61
LDA  $3Ø
STA  $62
LDY  #$ØØ
LDA  ($61),Y
CMP  #$DA
BNE  $CEØA
INY
LDA  ($61),Y
CMP  #$8Ø
BEQ  $CE12
LDY  #$Ø2
CLC
JSR  $CE6B
BCC  $CDFB
LDA  #$FF
LDY  #$Ø8
STA  ($61),Y
DEY
LDA  #$FF
STA  ($61),Y
SEC
JSR  $CE6B
LDA  #$Ø7
STA  $CD99
LDA  #$ØØ
STA  $CD42
STA  $CE4Ø
```

```
LDY   #$FF
LDA   ($2D),Y
STA   $CE42
INY
LDA   ($2D),Y
STA   $65
INY
LDA   ($2D),Y
STA   $66
LDY   #$FF
CPY   #$FF
BEQ   $CE4F
LDA   ($65),Y
JSR   $CD14
INC   $CE4Ø
BNE   $CE3F
JSR   $CD14
LDA   #$Ø6
CMP   $CD99
LDA   $61
SBC   $63
STA   $CE1A
LDA   $62
SBC   $64
STA   $CE13
PLA
TAY
PLA
TAX
PLA
RTS
LDA   $61
STA   $63
ADC   ($61),Y
STA   $61
INY
LDA   $62
STA   $64
ADC   ($63),Y
STA   $62
RTS
LDX   #$1F
PHA
SEC
SBC   $CE7D,X
BEQ   $CD3F
CMP   #$8Ø
BEQ   $CD25
CMP   #$2Ø
BNE   $CD3Ø
TXA
PHA
LDA   #$5E
JSR   $CD14
PLA
TAX
BPL   $CD3F
CMP   #$EØ
BNE   $CD3A
TXA
PHA
LDA   #$5F
BNE   $CD29
PLA
DEX
BPL   $CD16
RTS
PLA
TXA
LDY   #$FF
CMP   #$ØA
BEQ   $CD4A
STA   $CD42
LDX   $CD99
DEX
DEX
CMP   $CE9D,Y
BNE   $CD58
LDA   #$ØØ
BEQ   $CD89
DEX
CMP   $CEBE,Y
BNE   $CD62
LDA   #$4Ø
BNE   $CD89
CMP   #$Ø1
BPL   $CD6A
LDA   #$6Ø
BPL   $CD89
DEX
DEX
DEX
DEX
DEX
ADC   #$DF
CMP   #$F8
BPL   $CD89
INX
ASL   A
ADC   #$Ø7
CMP   #$FØ
BPL   $CD89
INX
ASL   A
ADC   #$ØF
CMP   #$C8
BPL   $CD89
INX
ASL   A
ADC   #$37
CPY   #$ØB
BNE   $CD92
LDY   #$2Ø
STY   $CD42
STX   $CDB5
LDX   $CD99
BNE   $CD9A
ROR   A
ROR   A
ROR   A
ROR   A
ROR   A
ROR   A
ROR   A
ROR   A
PHA
ROL   A
AND   $CEEØ,X
LDY   #$ØØ
ORA   ($61),Y
STA   ($61),Y
INY
PLA
AND   $CEE8,X
STA   ($61),Y
LDA   =$FF
BPL   $CDCØ
EOR   #$F8
INC   $61
BNE   $CDCØ
INC   $62
STA   $CD99
LDA   #$ØØ
RTS
PHA
```

The BBC and Electron both have their own built-in assemblers, so there will be no hex listing published next week. The first part of the text compressor follows. Type it in and SAVE it, but don't RUN—assemble—it yet because the second half will follow next time.

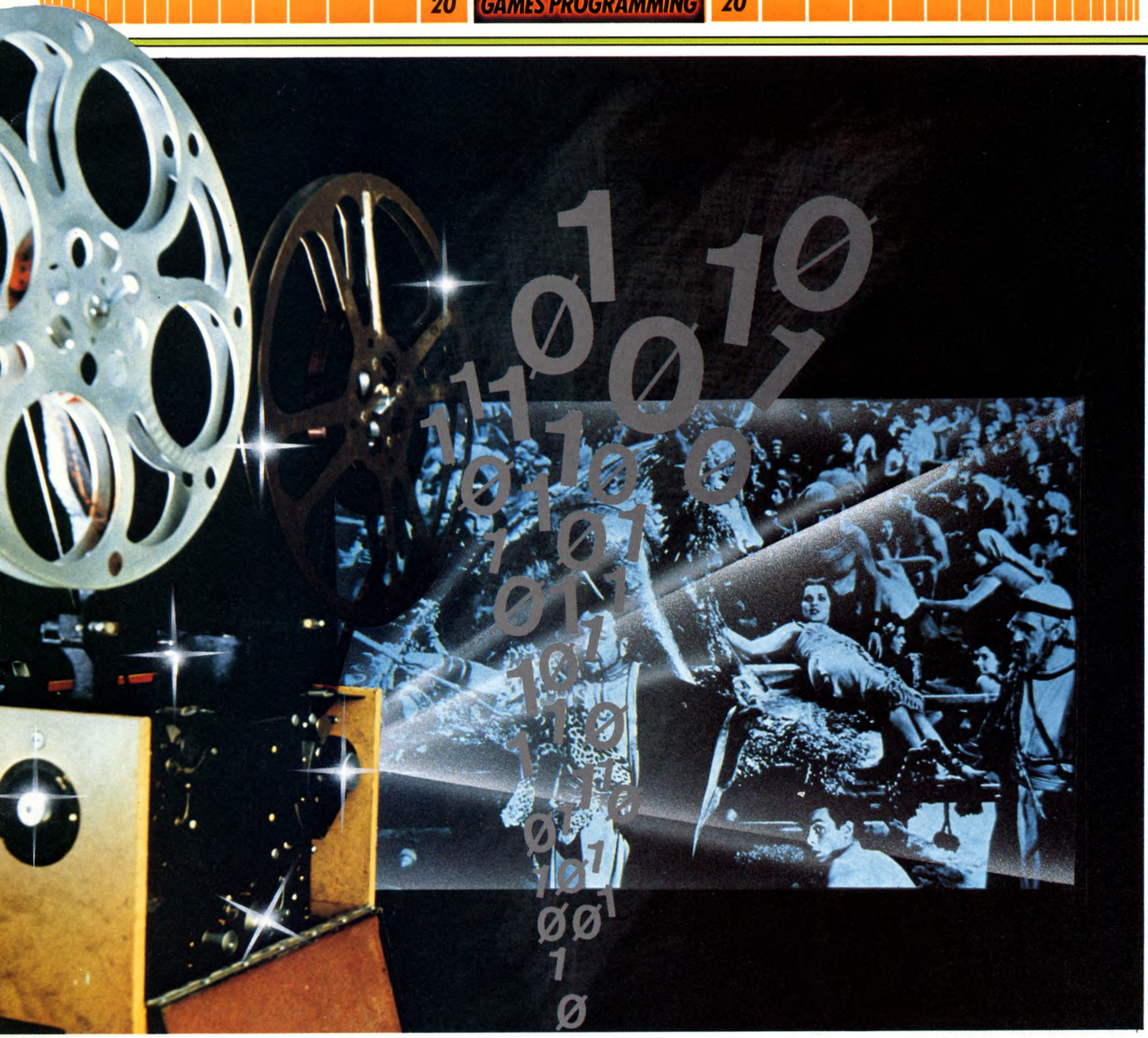
```
10 MODE6
20 HIMEM=&6ØØØ-&2ØØ
30 MC=HIMEM
40 P%=MC
50 ZCOD=112:ZPCT=114:ZDOL=116:
   ZBOX=118:ZUSE=116
60 CUP=P%+1Ø:CAR=P%+11:
   CLAST=P%+31:FIRST=P%+32:
   SECOND=P%+65:LO=P%+98:
   UP=P%+1Ø6:CPUN=FIRST
70 CODE=P%:$P%="□ASOTRILEC":
   P%=P%+1Ø
80 CSIX=P%:?P%=&5E:P%?1=&5F:
   $(P%+2)="UMPWYNBG":
   P%=P%+1Ø
90 CSEVEN=P%:$P%="DFVH":
   P%=P%+4
100 CEIGHT=P%:$P%="KQXJZ[\]"
110 P%=FIRST
120 FOR T=1 TO 2
130 READ A$
140 FOR P=1 TO LEN(A$) STEP 2
150 V=EVAL("&"+MID$(A$,P,2))
160 IF V<16 THEN PRINT"Ø";
170 PRINT;~V;
180 ?P%=V:P%=P%+1
190 NEXT
200 NEXT
210 FOR T=Ø TO 3 STEP 3
220 P%=MC+&74
230 [OPT T
240 .SETUP□LDA #&Ø4: LDX #&F4
250 .NXTZ□STX ZUSE: STA ZUSE+1
260 LDY #Ø2: LDA (ZUSE),Y
270 CMP #ASC("$"): BNE NTDOL
280 STX ZBOX: LDA ZUSE+1
290 STA ZBOX+1: BNE DON
300 .NTDOL□CMP #ASC("%")
310 BNE DON: STX ZPCT
320 LDA ZUSE+1: STA ZPCT+1
330 .DON□LDY #Ø: LDA (ZUSE),Y
340 TAX: INY: LDA (ZUSE),Y
350 BNE NXTZ: .OUTZ□RTS
360 .ESETUP□LDY #12
370 STY ESTRING+1
380 LDY #Ø: STY HIOFF+1
390 JMP SETUP
400 .ESTRING□LDA #&FF: PHA
410 ADC ZPCT: STA ZCOD
420 .HIOFF□LDA #&FF: PHA
430 ADC ZPCT+1: STA ZCOD+1
440 LDY #7: STY JAR+1: LDA #Ø
450 STA PREFCD+1
460 STA EASTER+1
470 LDA (ZBOX),Y
480 STA LCNPT+1: LDY #5
490 LDA (ZBOX),Y: STA ZDOL+1
500 DEY: LDA (ZBOX),Y
510 STA ZDOL
520 .EASTER□LDY #&FF
530 .LCNPT□CPY #&FF
540 BEQ EEXIT: LDA (ZDOL),Y
550 JSR ENCODE: INC EASTER+1
560 BNE EASTER
570 .EEXIT□JSR ENCODE
580 LDA JAR+1: CMP #7
590 JSR ZINC: SEC: LDA ZCOD
600 SBC ZPCT: STA ESTRING+1
610 LDA ZCOD+1: SBC ZPCT+1
620 STA HIOFF+1: PLA: TAX: PLA
630 RTS
640 .ENCODE□CMP #ASC("@")
650 BMI ENC2: EOR #&2Ø
660 .ENC2□LDX #CLAST-CODE
670 .TRYC□PHA: SEC: SBC CODE,X
680 BEQ MATCH: CMP #&2Ø
690 BNE NTUPP: TXA: PHA
700 LDA #&5E: .REDO□JSR ENC2
710 PLA: TAX: BPL MATCH
720 .NTUPP□CMP #&EØ
730 BNE NTLOW: TXA: PHA
740 LDA #&5F: BNE REDO
750 .NTLOW□PLA: DEX: BPL TRYC
760 RTS: .MATCH□PLA: TXA
770 .PREFCD□LDY #&FF
780 CMP #CUP-CODE
790 BEQ BITCNT: STA PREFCD+1
800 .BITCNT□LDX JAR+1: DEX
810 DEX: CMP FIRST,Y
820 BNE NTFIR: LDA #Ø: BEQ TP
830 .NTFIR□DEX: CMP SECOND,Y
840 BNE NTSEC: LDA #&4Ø
850 BNE TP: .NTSEC□CMP #1
860 BPL LONG: LDA #&6Ø: BPL TP
870 .LONG□DEX: DEX: DEX: DEX
880 DEX: ADC #254+CODE-CLAST
890 CMP #255+CEIGHT-CLAST
900 BPL TP: INX: ASL A
910 ADC #CLAST-CEIGHT
920 CMP #255+CSEVEN-CLAST+
   CSEVEN-CEIGHT
```

```
930 BPL TP: INX: ASL A
940 ADC #CLAST-CSEVEN+
  CEIGHT-CSEVEN
950 CMP #255+CSIX*4-2*
  CSEVEN-CEIGHT-CLAST
960 BPL TP: INX: ASL A
970 ADC #CLAST-CSIX*4+2*
  CSEVEN+CEIGHT
980 .TP□CPY #CAR-CODE
990 BNE TPACT
1000 LDY #CPUN-CODE
1010 STY PREFCD+1
1020 .TPACT□STX LADA+1
1030 LDX JAR+1
1040 .JAR□BNE JAR+2: ROR A
1050 ROR A: ROR A: ROR A
1060 ROR A: ROR A: ROR A
1070 ROR A: PHA: ROL A
1080 AND LO+1,X: LDY #0
1090 ORA (ZCOD),Y
1100 STA (ZCOD),Y: INY: PLA
1110 AND UP+1,X: STA (ZCOD),Y
1120 .LADA□LDA #&FF
1130 .ZINC□BPL DONE
1140 EOR #&F8: INC ZCOD
1160 BNE DONE: INC ZCOD+1
1170 .DONE□STA JAR+1: LDA #0: RTS
1180 ]:NEXT
1190 *SAVE"ENCODE" 5E00□5FC0
```

The Dragon and Tandy versions of the coder are almost identical, but for three small alterations. The things to alter are printed in bold type, and if you own a Tandy you should change them as follows: change 8B 30 to B3 ED and change 8C 37 to B4 F4. The program was written using the DASM assembler. If you have a different one, you may have to change the @ signs which denote labels. Apostrophes are used instead of exclamation marks on the Tandy assembler to denote ASCII code.

The assembly listing should be entered, assembled and then SAVEd to tape. The start address is 32380:

```
@CODE EQU *
@CSP FCB $20
@CA  FCB !A
@CS  FCB !S
@CO  FCB !O
@CT  FCB !T
@CR  FCB !R
@CI  FCB !I
@CL  FCB !L
@CE  FCB !E
     FCB !C
@CSIX EQU *
@CUP FCB !^
@CAR FCB $5F
@CU  FCB !U
     FCB !M
@CP  FCB !P
     FCB !W
     FCB !Y
@CN  FCB !N
     FCB !B
     FCB !G
@CSEVEN EQU *
@CD  FCB !D
@CF  FCB !F
     FCB !V
@CH  FCB !H
@CEIGHT EQU *
     FCB !K
     FCB !Q
@CX  FCB !X
     FCB !J
     FCB !Z
     FCB $5B
@CPO FCB $5C
@CLAST FCB $5D
@CPUN EQU *
@FIRST FCB @CUP-@CODE
     FCB @CN-@CODE
     FCB @CT-@CODE
     FCB @CN-@CODE
     FCB @CH-@CODE
     FCB @CE-@CODE
     FCB @CN-@CODE
     FCB @CE-@CODE
     FCB @CR-@CODE
     FCB @CH-@CODE
     FCB @CS-@CODE
     FCB @CL-@CODE
     FCB @CN-@CODE
     FCB @CE-@CODE
     FCB @CO-@CODE
     FCB @CA-@CODE
     FCB @CA-@CODE
     FCB @CD-@CODE
     FCB @CE-@CODE
     FCB @CE-@CODE
     FCB @CE-@CODE
     FCB @CT-@CODE
     FCB @CE-@CODE
     FCB @CE-@CODE
     FCB @CE-@CODE
     FCB @CU-@CODE
     FCB @CP-@CODE
     FCB @CU-@CODE
     FCB @CE-@CODE
     FCB @CAR-@CODE
     FCB @CAR-@CODE
     FCB @CSP-@CODE
     FCB @CAR-@CODE
@SECOND FCB @CT-@CODE
     FCB @CT-@CODE
     FCB @CE-@CODE
     FCB @CF-@CODE
     FCB @CI-@CODE
     FCB @CO-@CODE
     FCB @CT-@CODE
     FCB @CI-@CODE
     FCB @CD-@CODE
     FCB @CO-@CODE
     FCB @CT-@CODE
     FCB @CN-@CODE
     FCB @CS-@CODE
     FCB @CA-@CODE
     FCB @CE-@CODE
     FCB @CE-@CODE
     FCB @CH-@CODE
     FCB @CO-@CODE
     FCB @CT-@CODE
     FCB @CL-@CODE
     FCB @CH-@CODE
     FCB @CI-@CODE
     FCB @CO-@CODE
     FCB @CI-@CODE
     FCB @CA-@CODE
     FCB @CI-@CODE
     FCB @CX-@CODE
     FCB @CT-@CODE
     FCB @CO-@CODE
     FCB @CI-@CODE
     FCB @CUP-@CODE
     FCB @CPO-@CODE
     FCB @CAR-@CODE
     FCB @CUP-@CODE
@LO  FCB 0
     FCB $1
     FCB $3
     FCB $7
     FCB $F
     FCB $1F
     FCB $3F
     FCB $7F
@UP  FCB $FF
     FCB $FE
     FCB $FC
     FCB $F8
     FCB $F0
     FCB $E0
     FCB $C0
     FCB $80
     FCB 0
@USR9 PSHS D
     JSR $8B30
     STD @ZPTR+1
     CLR @MARRY+1
     CLR @MARRY+2
     PULS D
     RTS
@USR8 PSHS A,B,X,Y,U
     LDA #7
     STA @JAR+1
     LDD #@CSP
     STD @PREFCD+1
     JSR $8B30
     TFR D,X
     PSHS X
@MARRY LDD #$ABCD
```




```
        LEAY D,X
        JSR $8C37
@ZPTR LDU #$ABCD
        PULU B
        PULU A
        ADDD ,U
        LDU ,U
        STD @CAMP+2
        BRA @CAMP
@ENCLOP PULU A
        PSHS U
        JSR @ENCODE
        PULS U
@CAMP CMPU #$ABCD
        BNE @ENCLOP
        CLRA
        JSR @ENCODE
        CMPA #7
        BEQ @PULSE
        LDA ,Y+
@PULSE PULS D
        COMA
        COMB
        ADDD #1
        LEAX D,Y
        STX @MARRY+1
        PULS A,B,X,Y,U
        RTS
@ENCODE LDU #@CLAST+1
        TFR A,B
@TRYC SUBA ,-U
        BEQ @MATCH
        CMPA #$2Ø
        BNE @NOUPP
        LDA #!∧
@REDO PSHS U
        JSR @ENCODE
        PULS U
        BRA @MATCH
@NOUPP CMPA #$EØ
        BNE @NOLOW
        LDA #$5F
        BRA @REDO
@NOLOW TFR B,A
        CMPU #@CODE-1
        BGT @TRYC
        RTS
@MATCH TFR U,D
@PREFCD LDX #$ABCD
        LDA @JAR+1
        STA @EXTRA+2
```

```
        SUBA #2
        SUBB #@CODE (see note overleaf)
        BEQ @BITCNT
        CMPX #@CAR
        BNE @STEW
        LDU #@CPUN
@STEW STU @PREFCD+1
@BITCNT CMPB @FIRST-@CODE,X
        BNE @NOTFIR
        LDB #Ø
        BRA @TOP
@NOTFIR DECA
        CMPB @SECOND-@CODE,X
        BNE @LONG
        LDB #$4Ø
        BRA @TOP
@SPACEE LDB #$6Ø
        DECA
        BRA @TOP
@LONG SUBA #5
        SUBB #@CLAST-@CODE+1
        CMPB #255+@CEIGHT-@CLAST
        BPL @TOP
        INCA
        ASLB
        ADDB #@CLAST-@CEIGHT+1
        CMPB #255+@CSEVEN-
        @CLAST+@CSEVEN-@CEIGHT
        BPL@TOP
        INCA
        ASLB
        ADDB #@CLAST-@CSEVEN+
        @CEIGHT-@CSEVEN+1
        CMPB #255+@CSIX-@CLAST+
        @CSIX-@CEIGHT+@CSIX-
        @CSEVEN+@CSIX-@CSEVEN
        BPL @TOP
        INCA
        ASLB
        ADDB #@CLAST-@CSIX+
        @CEIGHT-@CSIX+@CSEVEN-
        @CSIX+@CSEVEN-@CSIX+1
@TOP EQU *
@EXTRA LDX #$ØØFF
        LSRB
@JAR BRA $F+*
        RORB
        RORB
        RORB
        RORB
        RORB
        RORB
        RORB
        PSHS B
        ROLB
        ANDB @LO+1,X
        ORB ,Y
        STB ,Y
        PULS B
        ANDB @UP+1,X
        STB 1,Y
        CMPA #$ØØ
        BGE @DONE
        ADDA #8
        LDB ,Y+
@DONE STA @JAR+1
        RTS
        END
```

Note: Most assemblers will show an error at SUBB#@CODE. Don't worry, the program should run. If not, replace @CODE with $86.

CODE VALUES

Main character	If in upper case	If in punctuation code	Most popular 1st (ØØ)	2nd (Ø1Ø)	Coded
space	@	message ends	↑	t	Ø11
a	A	!	n	t	1ØØØØ
b	B	"	e	l	111Ø1Ø
c	C	#	h	o	11ØØØ
d	D	$	e	i	1111ØØØ
e	E	%	r	d	1Ø111
f	F	&	t	o	1111ØØ1
g	G	'	e	h	111Ø11
h	H	(	e	a	1111Ø11
i	I	)	n	t	1Ø1Ø1
j	J	*	u	o	11111Ø11
k	K	+	e	i	11111ØØØ
l	L	,	e	i	1Ø11Ø
m	M	-	e	a	11Ø1Ø1
n	N	.	d	t	111ØØ1
o	O	/	n	f	1ØØ1Ø
p	P	Ø	o	e	11Ø11Ø
q	Q	1	u	x	11111ØØ1
r	R	2	e	o	1Ø1ØØ
s	S	3	t	e	1ØØØ1
t	T	4	h	i	1ØØ11
u	U	5	n	s	11Ø1ØØ
v	V	6	e	i	1111Ø1Ø
w	W	7	a	h	11Ø111
x	X	8	p	t	11111Ø1Ø
y	Y	9	a	o	111ØØØ
z	Z	:	e	i	111111ØØ
↑		>	s	t	11ØØ1Ø
←		?	,(l)	.(n)	11ØØ11

Main character	If in punctuation code	Most popular 1st (ØØ)	2nd (Ø1Ø)	Coded
Spectrum				
{	;	←	↑	111111Ø1
|	<	←	|	1111111Ø
}	=	space	←	11111111
Commodore/Vic				
⊞	;	←	↑	111111Ø1
▧	<	←	▧	1111111Ø
◫	=	space	←	11111111
Acorn				
{	;	←	↑	111111Ø1
|	<	←	|	1111111Ø
}	=	space	←	11111111
Dragon/Tandy				
	;	←	↑	111111Ø1
	<	←		1111111Ø
	=	space	←	11111111

The table above is really intended for the characters in punctuation code, the main characters are unlikely to be used.

In the main table, for the Dragon and Tandy, you should read upper case letters in the 'main character' column and reversed characters in the 'upper case' column. The reason for this is that upper case characters are the most commonly used.

Where up arrows and left arrows appear in the table, please note that these are not keyboard symbols, but are used by the machine to go into capital or punctuation mode. They will not work at all as keyboard symbols.

Note: when using the text compressor, use *only* characters from the table, or the text will become corrupted.

DISCOVER HOW EASY IT IS TO IMPROVE, ALTER AND REPAIR YOUR HOME, WHEN YOU'VE GOT THE KNACK.

This step-by-step guide gives you the confidence to tackle every DIY job – and succeed!

The Knack is all the DIY skills you've ever wanted and will ever need, in one of the most comprehensive series of books ever published on the subject.

Decorating, tiling, carpentry, plumbing, heating – you name it and you'll find the techniques of each skill clearly explained and fully illustrated.

Thousands of step-by-step colour photographs and diagrams will show you everything you need to know to get a professional finish, at a fraction of the cost, everytime you DIY.

NO STAMP REQUIRED

SPECIAL INTRODUCTORY OFFER

Simply address your envelope (no stamp required) to:
Marshall Cavendish House, FREEPOST, Hove, East Sussex, BN3 2ZZ.

YES! I'D LIKE TO TRY VOLUME 1 TO KEEP FREE WITH NO COMMITMENT TO BUY ANYTHING.

Please send me Volume 1 of **The Knack** for me to keep **FREE** along with Volume 2 on ten day trial. If not satisfied I shall return Volume 2 and cancel my order. Or I shall pay for Volume 2 and you will then send me the remaining Volumes (3-24) on ten day trial. For those I keep I need only pay for at the rate of ONE-VOLUME-A-MONTH. Each volume costs £5.95 (fully inclusive of postage and handling) – except of course **Volume 1 which is mine to keep FREE in any event.**

Signature .... **(I am over 18)**
Orders not accepted without signature

Name ....

Address ....

....

.... **Post Code** ....

Tel No .... **STD Code** ....
Offer applies UK and Northern Ireland only — All orders subject to acceptance

KB 92

TAKE VOLUME 1 FREE AND RECEIVE VOLUME 2 ON TEN DAY TRIAL

Simply fill in the Introductory Offer form and post today.

SEND NO MONEY NOW NO STAMP NEEDED

THIS IS NOT A BOOKCLUB

HOW THIS OFFER WORKS

1. Your **FREE** Volume 1 will be sent to you along with Volume 2 on ten day trial. If you're not entirely satisfied, simply return Volume 2 only and your application will be cancelled.

NOTHING FURTHER WILL BE SENT TO YOU.

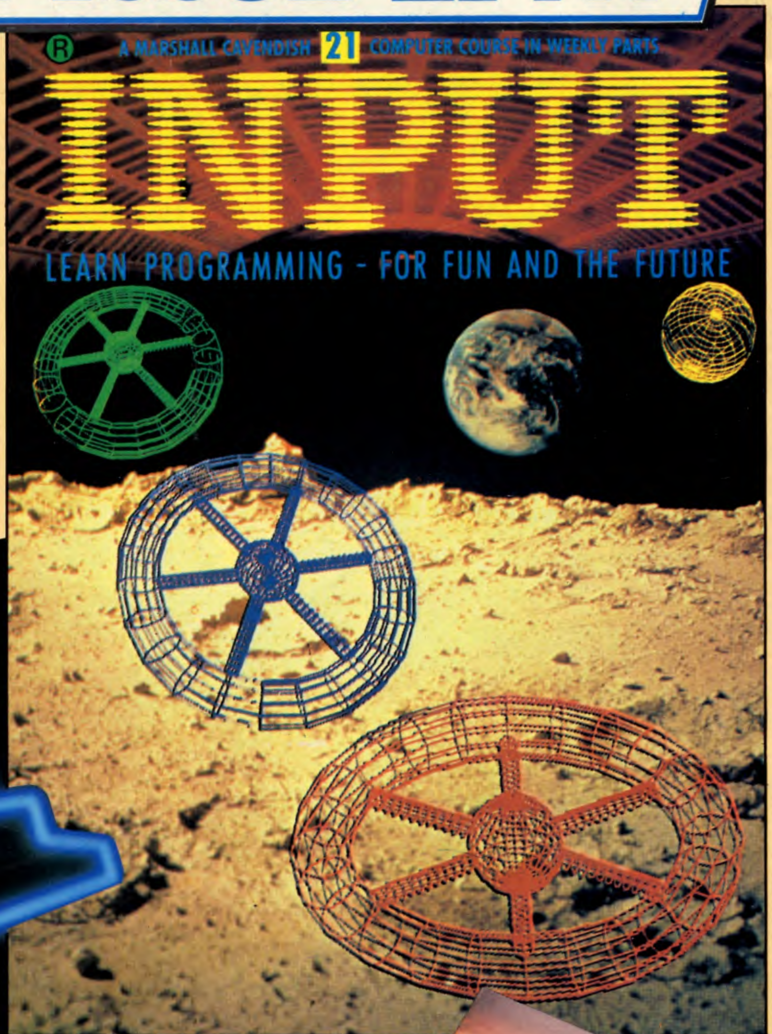
2. If you keep Volume 2 you pay just £5.95 (inclusive of postage and handling). Within a month you'll receive the further Volumes (3-24) all on ten day trial.

Even though you have the complete series, you only pay for ONE-VOLUME-A-MONTH which is just £5.95 (fully inclusive).

THIS IS NOT A BOOK CLUB – there is no commitment to buying a minimum number of books by accepting this FREE OFFER.

COMING IN ISSUE 21 ...

- ***Learn how to DETECT THINGS ON SCREEN—the basis of many games which involve moving graphics***
- ***Create a complete WIREFRAME picture of a SPACE STATION***
- ***Add the DECODING routine to your MACHINE CODE TEXT COMPRESSOR***
- ***Get it down on paper with a guide to SETTING UP A PRINTER***
- ***Plus, for the DRAGON and TANDY, a PROGRAM SQUEEZER to save memory***



ASK YOUR NEWSAGENT FOR INPUT