A MARSHALL CAVENDISH 39 COMPUTER COURSE IN WEEKLY PARTS

INPUT

LEARN PROGRAMMING - FOR FUN AND THE FUTURE

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

INPUT

Vol. 3 **No 39**

INDEX

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

PICTURE CREDITS
Front cover, Paul Chave. Pages 1209, 1210, 1211, 1212, 1215, Dave King. Pages 1214, 1215, Dave King. Pages 1216, 1217, 1218, 1219, 1220, 1221, Paddy Mounter. Pages 1222, 1223, 1224, 1225, 1228, 1229, Paul Chave. Pages 1230, 1231, 1232, 1233, 1234, 1235, 1236, Mohsen John Modaberi.

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

There are four binders each holding 13 issues.

HOW TO ORDER YOUR BINDERS

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

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

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

Malta: Binders are available from local newsagents.

BACK NUMBERS

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

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

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

COPIES BY POST

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

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

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

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

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

INPUT IS SPECIALLY DESIGNED FOR:

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

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

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

MODELLING: FOOD FOR THOUGHT

- SIMULATING THE WEATHER
- NORMALLY DISTRIBUTED VARIABLES
- TRADING CONDITIONS
- BOOK-KEEPING

Using a more efficient modelling method, enter programs and see how business principles apply to making a profit from managing a food stall

Modelling reality, on pages 1198 to 1203, showed the importance of computer simulations in today's Hi-tech society. You saw, also, how to generate different types of random variables to suit various events. The method for generating normally distributed variables is easy to understand, but it is inefficient, as the Normal Simulation program showed—a total of 15 random numbers were used to generate a single *normal* variable. The program listed here employs a more efficient method which, together with some of the ideas developed in the earlier article, helps to model certain aspects of a small business. You can treat this program as a game (and it is an interesting one), but not before you have considered it in its true light—as a model of a real venture.

```
10 POKE 23658,8: POKE 23609,12
20 PAPER 0:BORDER 0:INK 7:CLS
30 PRINT AT 0,8;INVERSE 1;
   "□POTATOMANIA□"
40 DIM A$(4,12):DIM L(2):DIM D(10):DIM
   W(2):DIM Q(2):DIM P(2)
50 LET A$(1)="HOT AND DRY":LET
   D(3)=150:LET D(4)=300
60 LET A$(2)="HOT AND WET":LET D(5)=
   100:LET D(6)=200
70 LET A$(3)="COLD AND DRY":LET
   D(7)=250:LET D(8)=160
80 LET A$(4)="COLD AND WET":LET
   D(9)=200:LET D(10)=100
580 DIM C(4,2): LET C(1,1)=.1: LET
   C(1,2)=.15
590 LETC(2,1)=0.5:LETC(2,2)=0.25
600 LETC(3,1)=0.01:LETC(3,2)=0.12
610 LETC(4,1)=10LETC(4,2)=10
620 INPUT "HOW MANY PLAYERS
   (1-6)?□";N
625 IF N<1 OR N>6 THEN GOTO 620
627 DIM K(N): DIM T(2,N): DIM O(2,N)
630 FOR I=1 TO 2: FOR J=1 TO N
640 LET T(I,J)=0
```

```
650 NEXT J: NEXT I
700 FOR K=1 TO 10
710 LET P1=INT (10*(.3+(RND*1/2)))/10
720 LET P2=INT (10*(.2+(RND*1/2)))/10
730 PRINT INK 4; BRIGHT 1;"- - - - -
   - - - - - - - - - - - - - - - - -
   - - - - - - - - - - - - □□□Day:";
   INK 7;K;INK 4;"- - - - - - - - - - -
   - - - - - - - - - - - - - - - - -
   - - - - - - - - - -"''
740 PRINT "Prob. of a hot dry day
   is ";100*P1;"%"
750 PRINT "Prob. of a dry day
   is ";100*P2;"%"
760 GOSUB 1400
770 LET U1=RND*1: LET U2=RND*1
780 LET V1=SQR (2*(LN (1/U1)))
790 LET V2=COS (2*PI*U2): LET V3=SIN
   (2*PI*U2)
800 LET Z1=INT (V1*V2): LET Z2=INT
   (V1*V3)
810 LET A1=P1*P2: LET A2=P1
820 LET A3=P1+P2-A1: LET A4=1: LET
   F=RND*1
821 PRINT : IF F<A1
   THEN LET R=1
822 IF F>A1 AND F<=A2
   THEN LET R=2
823 IF F>A2 AND F<=A3
   THEN LET R=3
824 IF F>A3 AND F<=A4
   THEN LET R=4
830 CLS : PRINT "THE WEATHER
   IS□";A$(R)
840 LET D(1)=INT (D(1+R*2)+
   Z1*25): LET D(2)=INT
   (D(2+R*2)+Z2*40)
850 PRINT "Demand for baked
   potatoes=";D(1)
860 PRINT "Demand for cans of
   cola=";D(2)
990 PRINT '
1000 PRINT INK 5; INVERSE 1;" PLAYER
   TAKINGS COSTS PROFIT"
1010 GOSUB 1600
1020 NEXT K
1030 PAUSE 200
1090 CLS
1100 PRINT "FINAL RESULTS FOR 10
   DAYS"''
1110 PRINT "PLAYER","TOTAL PROFIT"
1120 FOR J=1 TO N
1130 PRINT J,K(J): NEXT J
1140 PRINT '': PRINT "GAME OVER"
1150 STOP
1400 PRINT "ORDERS PLEASE": PRINT
1410 FOR J=1 TO N
1420 PRINT "PLAYER ";J: PRINT
1430 INPUT "NUMBER OF HOT
   POTATOES ";O(1,J)
1440 INPUT "NUMBER OF COLA
```


```
   CANS ";O(2,J)
1450 NEXT J
1460 RETURN
1600 FOR J=1 TO N
1610 FOR I=1 TO 2
1620 LET L=O(I,J)
1630 IF D(I)<L THEN LET L=D(I)
1650 LET W(I)=C(2,I)*L
1670 LET Q(I)=C(1,I)*O(I,J)
1680 IF D(I)>L(I) THEN GOTO 1700
1690 LET Q(I)=Q(I)-C(3,I)*(O
   (I,J)-D(I))
1700 LET P(I)=W(I)-Q(I)
1710 LET T(I,J)=T(I,J)+P(I)
1720 NEXT I
1730 LET K(J)=T(1,J)+T(2,J)-200
1740 LET E=W(1)+W(2)
1750 LET C=Q(1)+Q(2)+20
1760 LET P=P(1)+p(2)-20
1770 PRINT INK 6;TAB 3;J;TAB 9;E;TAB
   18;C;TAB 25;P;'
1780 NEXT J
1790 RETURN
```

To run this program on the Vic 20, you need to fit a 3K memory expansion cartridge.

```
10 PRINT "■■>■■POTATOMANIA■
   ■"
20 DIM D$(10)
30 A$(1)="HOT AND DRY":D(3)=150:
   D(4)=300
40 A$(2)="HOT AND WET":D(5)=100:
   D(6)=200
50 A$(3)="COLD AND DRY":D(7)=250:
   D(8)=160
60 A$(4)="COLD AND WET":D(9)=200:
   D(10)=100
580 C1(1)=.1:C1(2)=.15
590 C2(1)=.5:C2(2)=.25
600 C3(1)=.01:C3(2)=.12
620 PRINT"HOW MANY PLAYERS (1-6)?"
625 GET N$:N=VAL(N$):IF N<1 OR N>6
   THEN 625
630 FOR I=1TO2:FOR J=1 TO N
640 TP(I,J)=0
```

```
650 NEXT J,I
700 FOR K=1 TO 10
710 P1=.1*INT(10*(.3+RND(1)/2))
720 P2=.1*INT(10*(.2+RND(1)/2))
730 PRINT "■■>■■DAY",K"■■"
740 PRINT "PROBABILITY OF A HOT":PRINT
    "DAY IS";100*P1;"%"
750 PRINT "PROBABILITY OF A DRY":PRINT
    "DAY IS";100*P2;"%"
760 GOSUB 1400
770 U1=RND(1):U2=RND(1)
780 V1=SQR(2*LOG(1/U1))
790 V2=COS(2*π*U2):V3=SIN(2*π*U2)
800 Z1=INT(V1*V2):Z2=INT(V1*V3)
810 A1=P1*P2:A2=P1
820 A3=P1+P2-A1:A4=1:F=RND(1)
821 IF F<=A1 THEN R=1
822 IF F>A1 AND F<=A2 THEN R=2
823 IF F>A2 AND F<=A3 THEN R=3
824 IF F>A3 AND F<=A4 THEN R=4
830 PRINT "WEATHER IS□";A$(R)
840 D(1)=INT(D(1+R*2)+Z1*25):D(2)=
    INT(D(2+R*2)+Z2*40)
850 PRINT "DEMAND FOR BAKED":PRINT
    "POTATOES IS";D(1)
860 PRINT "DEMAND FOR CANS OF":PRINT
    "COLA IS";D(2)
1000 PRINT "■■PLAYER,TAK,COSTS,
    PROF□"
1010 GOSUB 1600
1020 NEXTK
1030 FOR I=1 TO 2000:NEXT I
1100 PRINT "■FINAL RESULTS FOR":
    PRINT "10 DAYS TRADING■"
1110 PRINT "■PLAYER■■■■■
    TOTAL PROFITS"
1120 FOR J=1 TO N
1130 PRINT J,TT(J):NEXT J
1150 END
1400 PRINT "■>■■ORDERS PLEASE
    ■■"
1410 FOR J=1 TO N
1420 PRINT "PLAYER";J
1430 PRINT "■NUMBER OF HOT POTATOES
    REQUIRED":INPUT O(1,J)
1440 PRINT "NUMBER OF COLA CANS
    REQUIRED":INPUT O(2,J)
1450 NEXT J:PRINT"■"
1460 RETURN
1600 FOR J=1 TO N
1610 FOR I=1 TO 2
1620 L=O(I,J)
1630 IF D(I)<L THEN L=D(I)
1650 RV(I)=C2(I)*L
1670 TC(I)=C1(I)*O(I,J)
1680 IF D(I)<=L THEN TC(I)=TC(I)-C3
    (I)*(O(I,J)-D(I))
1700 P(I)=RV(I)-TC(I)
1710 TP(I,J)=TP(I,J)+P(I)
1720 NEXT I
1730 TT(J)=TP(1,J)+TP(2,J)-200
1740 E=RV(1)+RV(2)
1750 C=TC(1)+TC(2)+20
1760 P=P(1)+P(2)-20
1770 PRINT "■"J"■,"E",""C",""P
1780 NEXT J
1785 POKE 198,0:WAIT 198,1:POKE 198,0
1790 RETURN
```

```
20 DIM C1(2),C2(2),C3(2),D(10),O(2,6),P(2),
    RV(2),TT(6),TC(2),TP(2,6),A$(4)
30 A$(1)="HOT AND DRY":D(3)=150:
    D(4)=300
40 A$(2)="HOT AND WET":D(5)=100:
    D(6)=200
50 A$(3)="COLD AND DRY":D(7)=250:
    D(8)=160
60 A$(4)="COLD AND WET":D(9)=200:
    D(10)=100
575 MODE4
580 C1(1)=.1:C1(2)=.15
590 C2(1)=.5:C2(2)=.25
600 C3(1)=.01:C3(2)=.12
610 PRINT "POTATOMANIA"
620 PRINT"HOW MANY PLAYERS(1-6)":
    REPEAT N=GET-48:UNTIL (N>0 AND
    N<7)
630 FOR I=1 TO 2:FOR J=1 TO N
640 TP(I,J)=0
650 NEXT,
700 FOR K=1 TO 10
710 P1=.1*INT(10*(.3+RND(1)/2))
720 P2=.1*INT(10*(.2+RND(1)/2))
725 PRINTSTRING$(40,"_")
730 PRINT''"DAY□";K'
740 PRINT"PROBABILITY OF A HOT DAY
    IS□";100*P1;"%"
750 PRINT"PROBABILITY OF A DRY DAY
    IS□":;100*P2;"%"
760 GOSUB 1400
770 U1=RND(1):U2=RND(1)
780 V1=SQR(2*LN(1/U1))
790 V2=COS(2*PI*U2):V3=SIN(2*PI*U2)
800 Z1=V1*V2:Z2=V1*V3
810 A1=P1*P2:A2=P1
820 A3=P1+P2-A1:A4=1:F=RND(1)
821 IF F<=A1 THEN R=1
822 IF F>A1 AND F<=A2 THEN R=2
823 IF F>A2 AND F<=A3 THEN R=3
824 IF F>A3 AND F<=A4 THEN R=4
830 CLS:PRINT"THE WEATHER IS□";A$(R)
840 D(1)=INT(D(1+R*2)+Z1*25):
    D(2)=INT(D(2+R*2)+Z2*40)
850 PRINT"DEMAND FOR BAKED POTATOES
    IS□";D(1)
860 PRINT"DEMAND FOR CANS OF COLA
    IS□";D(2)
1000 PRINT''"PLAYER","TAKINGS",
    "COSTS","PROFIT"
1010 GOSUB 1600
1020 NEXT
1030 I=INKEY(200)
1090 CLS
1100 PRINT''"FINAL RESULTS FOR 10 DAYS
    TRADING"''
1110 PRINT"PLAYER", "TOTAL PROFITS"
1120 FOR J=1 TO N
1130 PRINT;J,TT(J):NEXT
1150 END
1400 PRINT'TAB(13)"ORDERS PLEASE"''
1410 FOR J=1 TO N
1420 PRINT"PLAYER□";J'
1430 INPUT"NUMBER OF HOT POTATOES
    REQUIRED",O(1,J)
1440 INPUT"NUMBER OF COLA CANS
    REQUIRED",O(2,J)
1450 NEXT
1460 RETURN
1600 FOR J=1 TO N
1610 FOR I=1 TO 2
1620 L=O(I,J)
1630 IF D(I)<L THEN L=D(I)
1650 RV(I)=C2(I)*L
1670 TC(I)=C1(I)*O(I,J)
```

```
1680 IF D(I)<=L THEN TC(I)=
   TC(I)-C3(I)*(O(I,J)-D(I))
1700 P(I)=RV(I)-TC(I)
1710 TP(I,J)=TP(I,J)+P(I)
1720 NEXT I
1730 TT(J)=TP(1,J)+TP(2,J)-200
1740 E=RV(1)+RV(2)
1750 C=TC(1)+TC(2)+20
1760 P=P(1)+P(2)-20
1770 PRINT;J,;E,;C,;P
1780 NEXT
1790 RETURN
```

```
10 PI=4*ATN(1)
20 CLS
30 PRINT@10,"potatomania":PRINT:PRINT
580 C1(1)=.1:C1(2)=.15
590 C2(1)=.5:C2(2)=.25
600 C3(1)=.01:C3(2)=.12
620 INPUT" HOW MANY PLAYERS(1-6)□";N
625 IF N<1 OR N>6 THEN 620
630 FOR I=1 TO 2:FOR J=1 TO N
640 TP(I,J)=0
650 NEXTJ,I
700 FOR K=1 TO 10
710 P1=INT(10*(.3+RND(0)/2))/10
720 P2=INT(10*(.2+RND(0)/2))/10
725 PRINT"PRESS ANY KEY TO CONTINUE"
726 IF INKEY$="" THEN 726
730 PRINT"day";K:PRINT
740 PRINT"PROB. OF A HOT DAY
   IS";100*P1;"%"
750 PRINT"PROB. OF A DRY DAY
   IS";100*P2;"%"
760 GOSUB1400
770 U1=RND(0):U2=RND(0)
780 V1=SQR(2*(LOG(1/U1)))
790 V2=COS(2*PI*U2):V3=SIN(2*PI*U2)
800 Z1=INT(V1*V2):Z2=INT(V1*V3)
810 A1=P1*P2:A2=P1
820 A3=P1+P2-A1:A4=1:F=RND(0)
821 CLS: IF F<=A1 THEN 830
822 IF F>A1 AND F<=A2 THEN 870
823 IF F>A2 AND F<=A3 THEN 810
824 IF F>A3 AND F<=A4 THEN 950
830 PRINT"WEATHER IS HOT AND DRY"
840 D(1)=150+Z1*25:D(2)=300+Z2*40:
   GOTO970
870 PRINT"WEATHER IS HOT AND WET"
880 D(1)=100+Z1*25:D(2)=200+Z2*40:
   GOTO970
910 PRINT"WEATHER IS COLD AND DRY"
920 D(1)=250+Z1*25:D(2)=160+Z2*40:
   GOTO970
950 PRINT"WEATHER IS COLD AND WET"
960 D(1)=200+Z1*25:D(2)=100+Z2*40
970 PRINT"DEMAND FOR BAKED
   POTATOES=";D(1)
980 PRINT"DEMAND FOR CANS OF
   COLA=";D(2)
1000 PRINT" PLAYER□□TAKINGS□□
   COSTS□□PROFITS"
1010 GOSUB1600
1020 NEXT K
1030 FOR I=1 TO 2000:NEXT
1090 CLS
1100 PRINT"□FINAL RESULTS FOR 10
   DAYS":PRINT:PRINT
1110 PRINT"PLAYER","TOTAL PROFIT"
1120 FOR J=1 TO N
1130 PRINTJ,TT(J):NEXT J
1140 PRINT:PRINT:PRINT"
   GAME OVER"
1150 END
1400 PRINTTAB(8);"orders
   please":PRINT
1410 FOR J=1 TO N
1420 PRINT"PLAYER";J:PRINT
1430 INPUT"NUMBER OF HOT
   POTATOES";O(1,J)
1440 INPUT"NUMBER OF COLA
   CANS";O(2,J)
1450 NEXT J
1460 RETURN
1600 FOR J=1 TO N
1610 FOR I=1 TO 2
1620 L=O(I,J)
1630 IF D(I)<L THEN L=D(I)
1650 RV(I)=C2(I)*L
1670 TC(I)=C1(I)*O(I,J)
1680 IF D(I)<=L THEN TC(I)=TC(I)-
   C3(I)*(O(I,J)-D(I))
1700 P(I)=RV(I)-TC(I)
1710 TP(I,J)=TP(I,J)+P(I)
1720 NEXT I
1730 TT(J)=TP(1,J)+TP(2,J)-200
1740 E=RV(1)+RV(2)
1750 C=TC(1)+TC(2)+20
1760 P=P(1)+P(2)-20
1770 PRINTUSING"□□□#□□□□#
   ###.##□#####.##□#
   ##.##";J,E,C,P
1780 NEXT J
1790 RETURN
```

The program concentrates on the 'sink-or-swim' aspect of a business—profit and loss. When you RUN, you can choose either to be the only participant or to trade alongside as many as five other managers. You might even choose to play the part of two or even three managers, making different decisions in each role. Take this as an opportunity to compare the results of, say, trading cautiously in one instance and enterprisingly, taking chances in another. Whatever you choose, enter the number of players to start.

Each player manages a hot potato stall selling hot potatoes and cans of cola drink. Demand depends on the weather. If the weather is cold, the potatoes tend to sell easily. If the sun shines, then plenty of drinks will be sold. Unfortunately, the manager needs to buy stock on the evening before the next day's trading, and so doesn't know the prevailing weather conditions. Fortunately, there is a weather forecast—correct about 70 per cent of the time. After ten days buying and selling, the player with the biggest profit is the winner.

The first part of the program (up to Line 65Ø) sets variables for screen display and for the trading conditions. You pay rent at £20 per day. You buy potatoes at 10p each and sell them at 50p each. Cola costs you 15p a can and you sell them at 25p. Goods left over from each day's trading are disposed of at scrap value—1p for potatoes and 12p for cola. Each game lasts ten days.

The demand indicator for goods is as in the table below, and this is where one important element enters into the model. Naturally, the best days for hot potatoes are among the poorest for cold drinks, but there is sufficient spread of demand between the two items to enable managers to do a reasonable trade, whatever the weather. All this, however, depends on what you make of the weather forecast, bearing in mind that it is not reliable—as in reality. The probabilities of a hot day and a dry day are determined at Lines 71Ø and 72Ø, then used (Lines 81Ø to 824) to simulate the weather.

Lines 77Ø to 8ØØ use a sophisticated method for generating normally distributed random variables. These are used to simulate demand at Lines 84Ø. Line 77Ø generates two random variables (U_1 and U_2), but remember that these are not truly random. So they are processed within three mathematical formulae. At Line 78Ø, U_1 is inverted and squared, then its natural log is taken. Finally, the square root of the result is set to V_1. Line 79Ø sets V_2 to the cosine of the circumference of a circle of radius U_2, then sets V_3 to the sine of the same circumference. The variables V_1, V_2 and V_3 are further processed (Line 8ØØ) to give normally distributed variables Z_1 and Z_2.

Besides Lines 16ØØ to the end of the program, the rest of the program deals with the organization of data input and the printing of results.

When you play this game, you will need to watch every penny. The results can be agonizingly close—even after ten days' trading.

DEMAND INDICATOR

	RANGE OF DEMAND	
WEATHER	(potatoes)	(Coke)
HOT AND DRY	100–200	220–380
HOT AND WET	50–150	120–280
COLD AND DRY	200–300	80–240
COLD AND WET	150–250	20–180

COMMODORE ASSEMBLER UPDATE

Enter these routines and turn your Commodore assembler into a far more useful tool. Now you can SAVE machine code to tape or disk and extend the existing features

The Commodore assembler on pages 402 to 405 does its job very well, but you may have wished for more facilities to help you develop your programs. This article contains a number of important additional features which will make machine code programming so much easier.

If you find using the machine code monitor to SAVE machine code rather fiddly and time-consuming, there is a complete machine code SAVE routine to add to the assembler. The program now has all you need to use either tape or disk for LOADing and SAVEing machine code.

THE SAVE ROUTINE

LOAD in your existing assembler—or, if you're starting from scratch, type in the program on pages 402 to 405, being careful to enter all the commas in the DATA lines—and add these lines:

```
185 PRINTQR$"π7◣□SAVE M-CODE"R$
    QR$"π8◣□CLEAR MEMORY"R$QR$
    "π9◣□EXIT PROGRAM"
1300 INPUT "▢▪START ADDRESS■";SA
1310 INPUT "▪□END ADDRESS□■";
    EN:EN=EN+1:JR=0
1320 NM$="":INPUT "▪□□FILE NAME
    □□■";NM$:IF NM$="" THEN 1320
1330 SZ=1:INPUT "▪(D)ISK,(T)APE■";
    D$:IF D$="D" THEN SZ=8
1333 IF D$<>"D" AND D$<>"T"
    THEN PRINT"▢";:GOTO 1330
1335 IF JR=1 THEN JR=0:RETURN
1340 S1=INT(SA/256):S2=SA-S1*256
1350 S3=INT(EN/256):S4=EN-S3*256
1360 PRINT"▢P□43,";S2;":P□44,";S1
1370 PRINT"▪▪P□45,";S4;":
    P□46,";S3
1380 PRINT"▪▪SAVE"+CHR$(34)+
    NM$+CHR$(34)+",";SZ;",1"
1390 PRINT"▪▪▪▪▪▪▪▪▪P
    □43,";PEEK(43);":P□44,";PEEK(44)
1400 PRINT"▪▪P□45,";PEEK(45);":P□
    46,";PEEK(46):PRINT"▪▪RETURN▤":
    END
1500 PRINTTAB(11)"▪ARE YOU SURE
    (Y/N)?"
1510 GETA$:IF A$="N"THEN RETURN
1520 IF A$="Y" AND JJ=8 THEN RUN
1530 IF A$="Y" AND JJ=9 THEN
    PRINT"▢◆":POKE 53280,14:POKE
    53281,6:END
1540 GOTO 1510
```

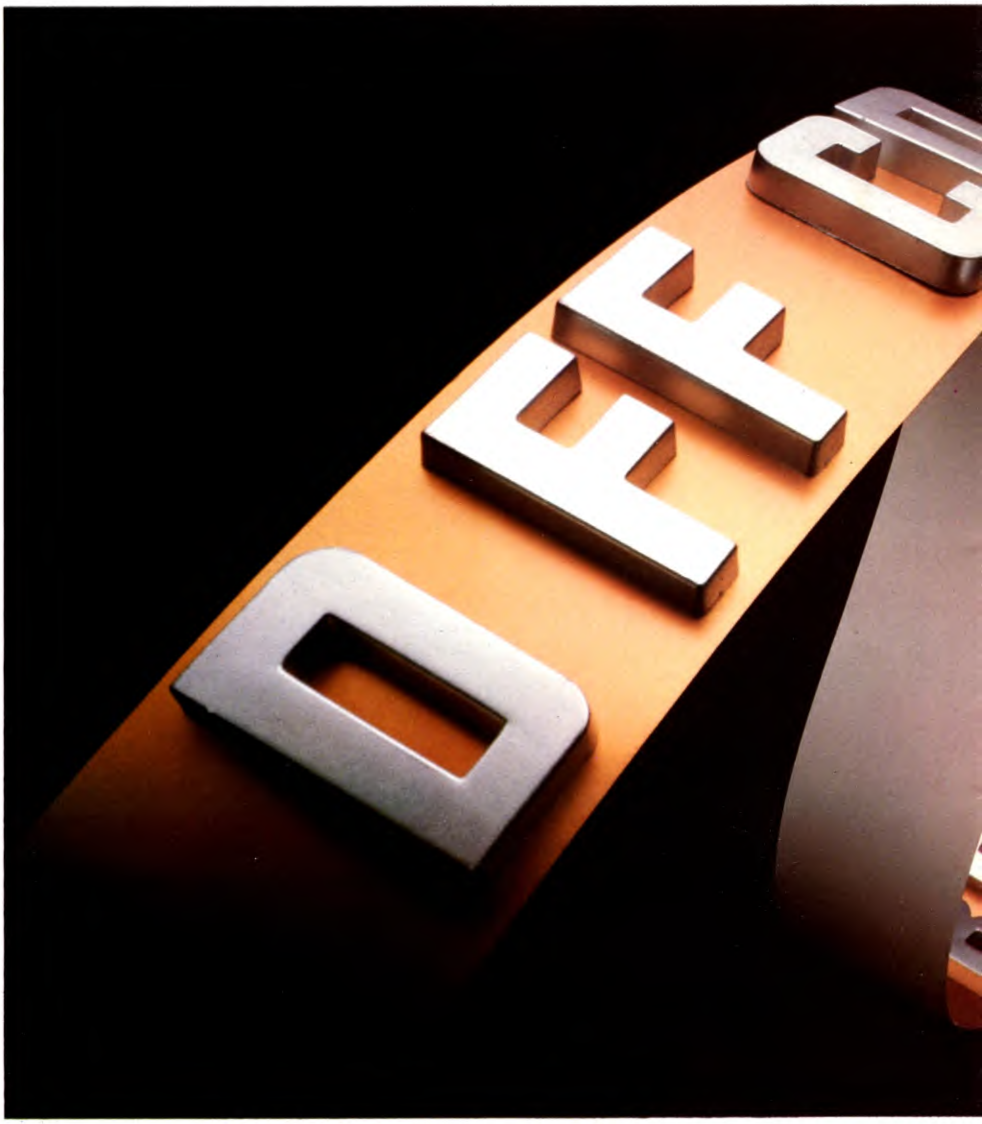
When you choose to SAVE the assembled code—option 7—you'll be asked for the start and end address of the code, a file name for the code to be SAVEd under and if you wish to SAVE to disk or tape.

The machine goes into direct mode, and displays two lists of POKEs. To SAVE machine code you should position the cursor at the first POKE, and hit RETURN three times.

The first three POKEs move BASIC so that the machine code can be SAVEd. Once SAVEing is completed, the second three POKEs move BASIC back again.

After SAVEing the machine code, you can clear the assembly language mnemonics from memory by choosing option 8.

If you wish to LOAD machine code, make sure you're outside the assembler. in other words, if the assembler is RUNning, you



- A MACHINE CODE SAVE FEATURE
- USING THE ASSEMBLER WITH TAPE OR DISK
- NAMING FILES
- OUT OF MEMORY WARNING
- IMPROVING THE EDIT MODE
- PRINTOUT FROM ASSEMBLING PROGRAMS
- A PAUSE FACILITY

should choose option nine to exit the program. Now type LOAD "name",1,1 (for tape) or LOAD "name",8,1 (for disk), and the code will LOAD.

MORE MODIFICATIONS

Options one and two can be modified, allowing you to LOAD assembly language from disk as well as tape. There is also an 'out of memory' feature, an improved edit mode, a facility for having printout from an assembling program, and a pause facility.

Here are the additions:

```
170 PRINTQR$"π1↑□LOAD ASSEM
   PROG"R$QR$"π2↑□SAVE ASSEM
   PROG"R$QR$"π3↑□ASSEMBLE"
200 GETA$:IFVAL(A$)<1ORVAL(A$)>9
   THEN200
220 ONJJGOSUB1Ø8Ø,11ØØ,25Ø,112Ø,12ØØ,
   123Ø,13ØØ,15ØØ,15ØØ
225 POKE198,Ø:PRINTTAB(15)"■■■■
   HIT ANY KEY"
255 INPUT "DO YOU WANT PRINTOUT
   (Y/N)";AN$:PRINT"♥":RK=Ø
28Ø IF PS=3 AND AN$="Y" AND RK=Ø
   THEN OPEN4,4:CMD4:RK=1
285 GOSUB 98Ø
295 IF PS=3 AND PEEK(197)=57 THEN
   POKE198,Ø:WAIT198,1:POKE198,Ø
31Ø IFOP$="END"ANDPS=3THEN
   PRINT"■END LAST ADDR";P-1:
   IFAN$="Y"THENPRINT#4:CLOSE4
1Ø8Ø JR=1:GOSUB 132Ø:OPEN1,SZ,Ø,NM$
11ØØ JR=1:GOSUB 132Ø:OPEN1,SZ,1,NM$
1135 IF N>199 THEN PRINT"MEMORY
   FULL!":RETURN
1145 IF IP$="↑" AND N>Ø THEN PRINT
   "□□":N=N-1:K=K-1Ø:GOTO
   113Ø
127Ø POKE198,Ø:PRINT"♥":FORK3=K1TO
   K2:PRINT"π"K3*1Ø"↑□"T$(K3)
128Ø WAIT 198,1:POKE 198,Ø:NEXT:RETURN
```

When you are editing an assembly language program, simply pressing the up arrow key and RETURN will move you back to the last line you edited, allowing easy correction of mistakes.

When the program is displaying assembled code, you can pause the listing by pressing the left arrow key. You can now note down any codes you may need to record before restarting the listing pressing any key.

USING THE ASSEMBLER

The program is menu-driven, and is very easy to use. However, there are a number of points you should bear in mind when using it:

- Make sure you have tested the program using the routine printed on page 4Ø4. If you have problems at this stage, check the assembler itself.
- If you suffer error messages arising from DATA lines, the most likely problem are the commas—for example, make sure you have included the comma at the end of Line 7Ø.
- Each separate assembly language mnemonic needs a BASIC line number. These should run 1Ø, 2Ø, 3Ø, 4Ø ..... and so on.
- Line 1Ø is normally an origin—such as 1Ø ORG 49152. The assembler will not work unless there is a space between ORG and the address.

CLIFFHANGER: THE RISING TIDE

Willie may be cute but he's no Canute and he is going to be threatened by the oncoming sea. In this part the rollers are set crashing and climbing

Willie is at the seaside and hardly a mention has been made of the sea yet. What's more, it is one of his major hazards. If he loses his nerve when the boulders are tumbling or the snakes are hissing, he will quickly find himself drowned. So now is the time to supply Cliffhanger with gallons of the briny.

The following routine turns on the flood tide so that Willie will not dawdle on the way to rescuing his picnic goodies:

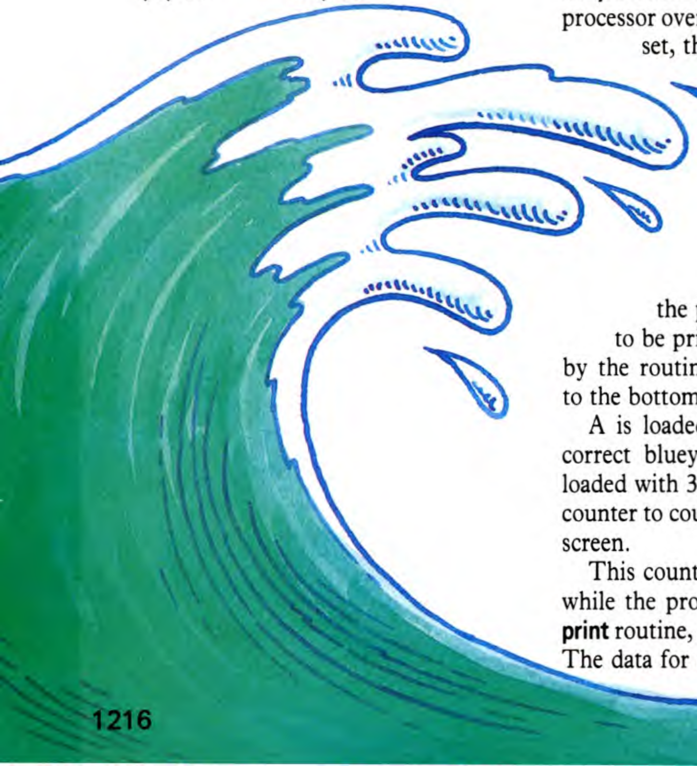
```
        org 58882
sea     ld bc,57312
        ld a,(57353)
        bit 2,a
        jr z,spt
        ld bc,5732Ø
spt     ld hl,(57354)
        ld a,15
        ld d,32
spu     push de
        push bc
        call print
        inc hl
        pop bc
        pop de
        dec d
        jr nz,spu
        ld a,(57353)
        dec a
        ld (57353),a
        jr nz,srt
        ld a,1Ø
        ld (57353),a
        ld hl,(57354)
        ld de,32
        sbc hl,de
        ld (57354),hl
srt     ret
        org 58217
print   *
```

Although there appears to be a lot of sea, there are, in fact, only two characters' worth in the data table. The first occupies the eight locations from 57,312 onwards and the second occupies the eight from 57,32Ø. You may think that this is not enough water even to dampen Willie's feet. But when these two sea characters are printed next to each other over and over again in alternate lines, you rapidly build up an ocean.

As the sea is going to be printed on the screen a character at a time, the print routine is going to be used again. So the relevant parameters have to be loaded into the correct registers. As always, BC carries the pointer to the first byte of data. A carries the colour. And HL carries the screen position the data is to be printed in. So BC is loaded up with the address of the first byte of the first sea character.

The variable in 57,353 is the so-called sea delay. This controls the movement of the sea. Bit two is used as a flag to tell the processor which sea character was used for the last line.

The contents of the sea delay are loaded into the accumulator and the instruction bit 2,a isolates that particular bit. If it is not set, the jr z instruction which follows it jumps the processor over the next instruction. But if it is set, the jump is not made, and BC is loaded with the address of the beginning of the second sea character.



SEA CHANGE

HL is loaded with the contents of memory location 57,32Ø. This location carries the position of the sea that is about to be printed and it has been initialized by the routine in part seven of Cliffhanger to the bottom left-hand corner of the screen.

A is loaded with 15, to give the sea the correct bluey tinge, and the D register is loaded with 32. This is going to be used as a counter to count across the 32 columns of the screen.

This counter needs to be preserved intact while the processor goes off to perform the **print** routine, so DE is pushed onto the stack. The data for the sea character is going to be used over again too, because each line of the sea is made up of the same sea character, so BC is pushed onto the stack as well. The **print** routine is then called, and the eight bytes of the appropriate sea character are printed up on the correct place on the screen.

HL is then incremented to move the screen pointer onto the next position along the row. The data pointer is moved back to the beginning of the appropriate sea character—it has been incremented during the **print** routine—by popping it off the stack. And the column counter is popped off the stack again too.

The column counter is then decremented and if it hasn't counted down to Ø, the jr nz instruction loops back so that the processor prints the sea character in the next screen position along that particular line.

When the counter in D has counted down to Ø, the processor drops out of the loop and proceeds with the next instruction.

TIME AND TIDE

The sea delay is then loaded into the accumulator, decremented and stored back into 57,353. If it has not counted down to zero, the **jr nz** instruction jumps the processor over the next few instructions to the end of the routine where it returns.

If it has counted down to zero, the sea delay is reset to 1Ø.

The sea position pointer is then loaded back into HL, 32 is loaded into DE and subtracted. The result in HL is stored back in the sea pointer's location, 57,354. So next time this routine is called, the sea is moved up one row.

This routine will move the sea up the screen by one line every time the sea counter variable in $CØØC is counted down to zero. And it scrolls the sea one high resolution pixel to the left to give the impression that the sea is moving.

```
        ORG 22272
        LDY #8
        LDX #Ø
LOOP    LDA $31FØ,X
        ROL A
        ROL $31FØ,X
        INX
        DEY
```

- THE SEA DELAY
- COLOURING THE SEA
- MOVING THE SEA UP
- REPEATING THE DROPS
- RESETTING THE SEA DELAY

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

```
        BNE LOOP
        DEC $C00C
        BEQ XX
        RTS
XX      LDA $C002
        STA $C00C
        LDY #0
        LDX #40
        LDA $C00D
        STA $FB
        LDA $C00E
        STA $FC
        SEC
        LDA $FB
        SBC #40
        STA $FB
        LDA $FC
        SBC #0
        STA $FC
        LDA $FB
        STA $C00D
        LDA $FC
        STA $C00E
        CLC
        ADC #212
        STA $FE
        LDA $FB
        STA $FD
LOOPA   LDA #82
        STA ($FB),Y
        LDA #6
        STA ($FD),Y
        INY
        DEX
        BNE LOOPA
        RTS
```

THE ROLLERS

The first part of this routine scrolls the sea to the left. But as one piece of sea looks very much like another, it is only necessary to rotate the bits of each byte on the screen, provided you don't lose the bit that goes into the carry flag. But this problem is got over simply.

Y is loaded with 8 as a counter—there are 8 bytes of data in each character square of the sea. X is loaded with 0 and is going to be used as an offset.

The first byte of sea data at $31F0 is loaded into the accumulator and rotated to the left. This puts its most significant bit into the carry flag. The data byte in $31F0 is then rotated as well, pulling the contents of the carry flag into its least significant bit. So the bits of the first byte of the sea data are rotated, without losing anything. This simple dodge has shifted what was in bit seven into bit zero.

The offset in X is then incremented to move onto the next byte and the loop counter in Y is decremented. If it has not counted down to 0, the processor loops back and rotates the next byte of sea data. But if all eight bytes have been rotated the processor drops out of the loop and proceeds with the next instruction.

MOVE ON UP

The contents of the sea counter in $C00C are decremented. And if they have not counted down to zero yet, it is not time to move the sea up, the BEQ condition allows the processor to pass through to the RTS and leave the routine.

But if it has counted down to zero, the processor skips the RTS and continues.

The sea counter is then reset by storing the delay variable in it. This allows the speed of the advance of the sea to be altered during the course of the game.

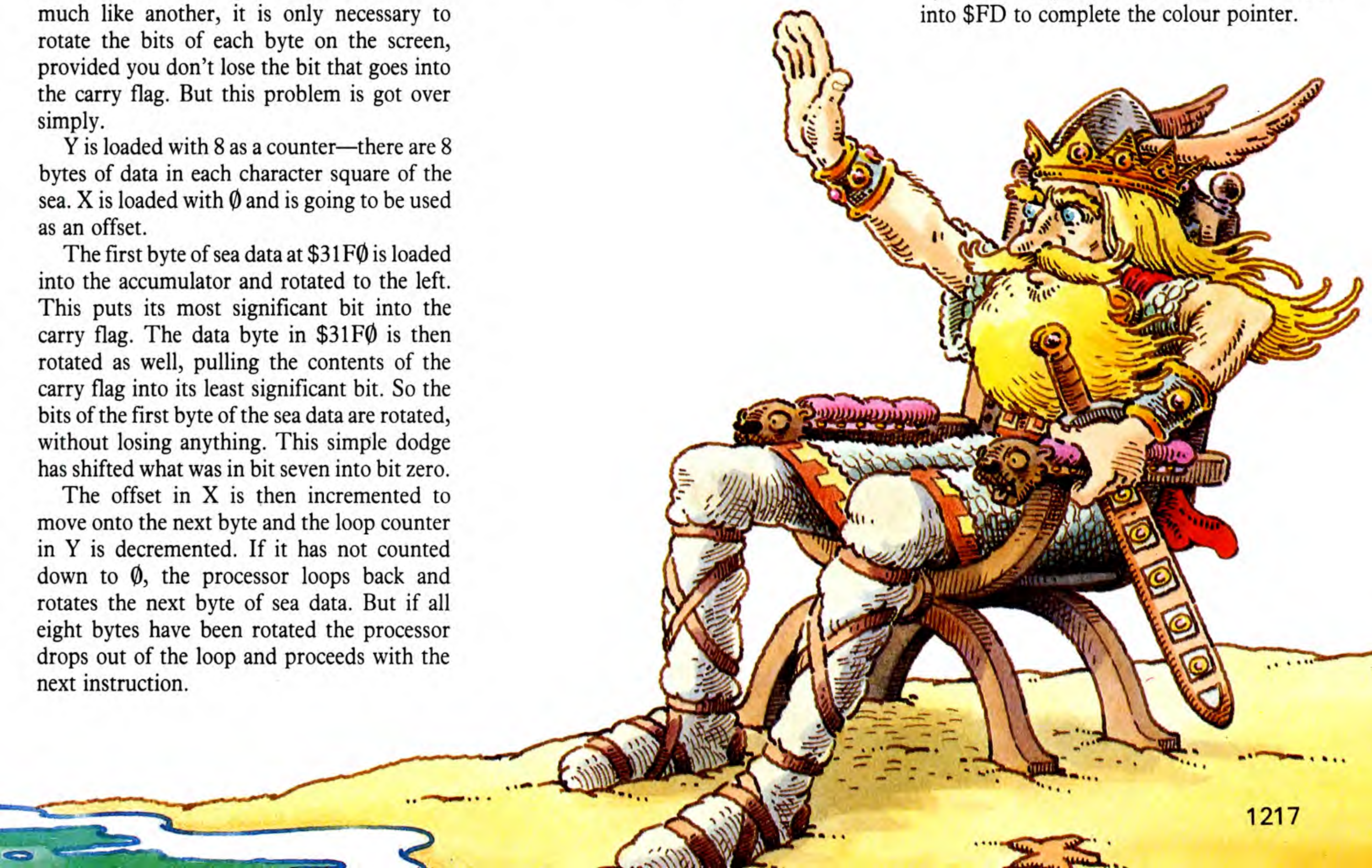
The Y register is loaded with 0. It is going to be used as an offset, as this time, pre-indexed indirect addressing is going to be used. Only post-indexed indirect addressing is allowed with the X register.

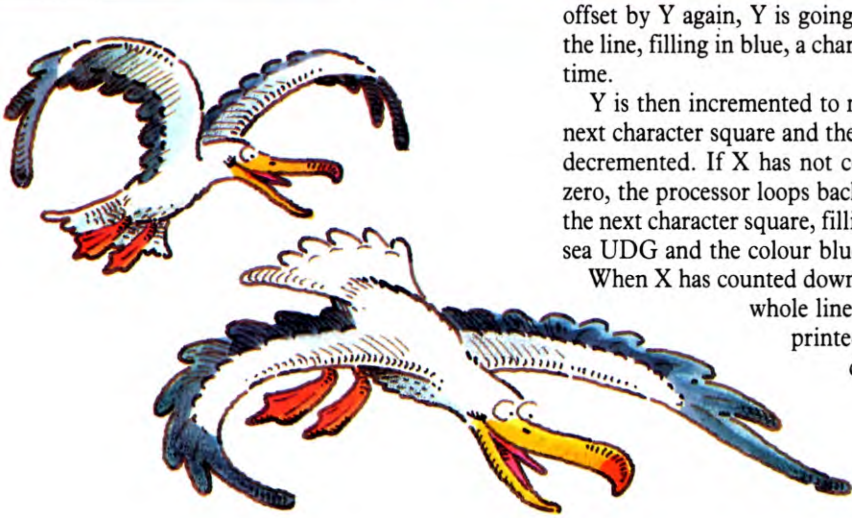
X is loaded with 40 to count across the 40-column screen.

The low and high bytes of the sea position variable in $C00D and $C00E are stored temporarily in the locations $FB and $FC on the zero page where thay can be manipulated easily. These point to the left-hand end of the line of sea about to be printed. The carry flag is then set—you are about to do a subtraction.

The number 40 is then subtracted from the low byte of the sea's screen position and zero is subtracted from the high byte to take any carry into account. This moves the sea position pointer one line up the screen. The result is stored back in $FB and $FC on the zero page and the pointer in the variable table at $C00D and $C00E.

That done, the carry flag is cleared and the number 212 is added to the high byte of the sea's screen position. This moves the pointer onto the appropriate location in the colour memory. The result is stored temporarily in zero-page memory location $FE. And the low byte of the sea position in $FB is copied direct into $FD to complete the colour pointer.





POURING IN THE SEA

UDG 62, which has been defined as a little bit of the sea in an earlier part of Cliffhanger, is loaded into the accumulator and stored on the screen in the position pointed to by the sea pointer in $FB and $FC, offset by Y—Y is going to count across the line, a character square at a time.

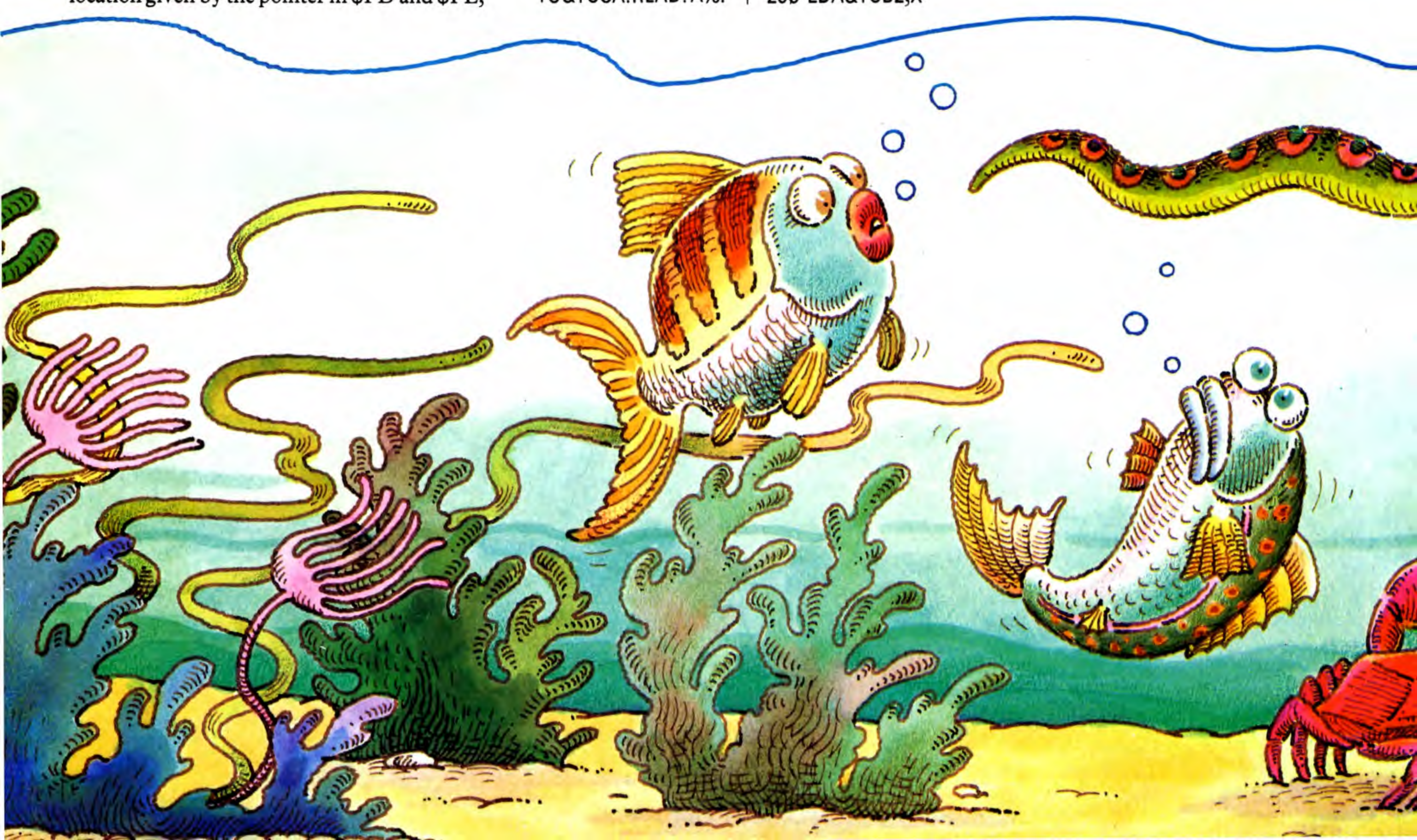
Then the accumulator is loaded with 6, the number associated with the colour blue. And the appropriate character square is filled in with blue, by storing the 6 colour memory location given by the pointer in $FD and $FE, offset by Y again, Y is going to count across the line, filling in blue, a character square at a time.

Y is then incremented to move it onto the next character square and the counter in X is decremented. If X has not counted down to zero, the processor loops back and deals with the next character square, filling it in with the sea UDG and the colour blue.

When X has counted down to zero and the whole line of sea has been printed, the processor drops out of the loop, executes the RTS and leaves the routine.

The following routine prints the sea on the screen and moves it. Set the machine up as usual before you type it in.

```
70 DATA6,7,14,15
80 FORA%=&1CB2TO
   &1CB5:READ?A%:
   NEXT
130 DATA25,1,0,5,0,
   0,18,0,5,25,0,0,
   251,4,0,25,17,0,5,
   0,0
140 FORA%=&1CB6
   TO&1CCA:READ?A%:
   NEXT
180 FORPASS=
   0TO3STEP3
190 P%=&1CCB
200 [OPTPASS
210 .Move Sea
220 LDA#19
230 JSR&FFEE
240 LDX&87
250 LDA&1CB2,X
260 JSR&FFEE
270 LDA#4
280 JSR&FFEE
290 LDA#0
300 JSR&FFEE
310 JSR&FFEE
320 JSR&FFEE
330 INC&87
340 LDA&87
350 AND#3
360 STA&87
370 LDA#19
380 JSR&FFEE
390 LDX&87
400 LDA&1CB2,X
410 JSR&FFEE
420 LDA#6
430 JSR&FFEE
440 LDA#0
450 JSR&FFEE
460 JSR&FFEE
470 JSR&FFEE
480 DEC&77
490 BEQLb2
500 RTS
510 .Lb2
520 LDA#25
530 JSR&FFEE
540 LDA#4
550 JSR&FFEE
560 LDA#0
570 JSR&FFEE
580 JSR&FFEE
590 LDA&88
600 ASLA
610 ROL&70
620 ASLA
630 ROL&70
640 JSR&FFEE
650 LDA&70
660 AND#3
670 JSR&FFEE
680 LDA#18
690 JSR&FFEE
700 LDA#0
710 JSR&FFEE
720 LDX&87
730 LDA&1CB2,X
740 JSR&FFEE
750 LDX#0
760 .Lb1
770 LDA&1CB6,X
780 JSR&FFEE
790 INX
800 CPX#21
810 BNELb1
820 INC&88
830 LDA#5
840 STA&77
850 RTS
860 ]NEXT
```



To test the routine, the rest of the program must be in memory. Then key in:

```
?&83=Ø:?&88=Ø:CALL &1B32:REPEAT
   CALL &1CCB: FOR A%=Ø TO 2ØØ:
   NEXT:UNTIL ?&88=24Ø
```

SEA CHANGE

The impression that the sea is moving is given by redefining colours 6, 7, 14 and 15—which are blue and cyan, after being redefined in part five of Cliffhanger—from blue to cyan and back again. The data for the colours to be changed are in the DATA in Line 7Ø, and they are read into a data table where the machine code program can access it by Line 8Ø.

A line of white dots is drawn along the top of the sea to represent surf. The DATA for this is in Line 13Ø and it is read into a data table by Line 14Ø.

The colours are redefined in exactly the same way as they were in part five of Cliffhanger (page 1037 of *INPUT*). Here, though, the offset is stored in zero-page memory location &87, because the colours are not being changed in a closed loop here and the number of the colour would be lost when the X register was used elsewhere.

The colour change facility is switched on by loading 19 into the accumulator and jumping to the subroutine at FFEE. This is the same as a VDU19 in BASIC. The colour to be changed is read in from the data table by the instruction in Line 25Ø. 1CB2 is the base address of the data table and the offset in X is loaded up from &87 in the Line before.

The colour loaded up in that part of the routine is then changed to colour 4—which is blue. Then another three parameters also have to be filled in with Øs. These are not used by the colour change routine, but Acorn have reserved them for future use.

The offset in &87 is incremented in Line 33Ø to move onto the next colour. Then it is loaded into the accumulator, ANDed with 3 and stored back in &87. This stops the offset being incremented to more than 3. There are only four colours so the offset only has to count from Ø to 3.

The next part of the routine—from Line 37Ø to 47Ø—changes the next colour to colour 6, which is cyan, in exactly the same way.

THE INCOMING TIDE

The variable stored in &77 is the so-called sea delay. It counts the number of times the sea colours are moved before a new line of sea has to be printed on the screen.

This is set in the initialization routine and is reset to 5 at the end of this routine. It is simply a device to stop the sea advancing too fast. Here the sea only advances once every five times the colours are changed.

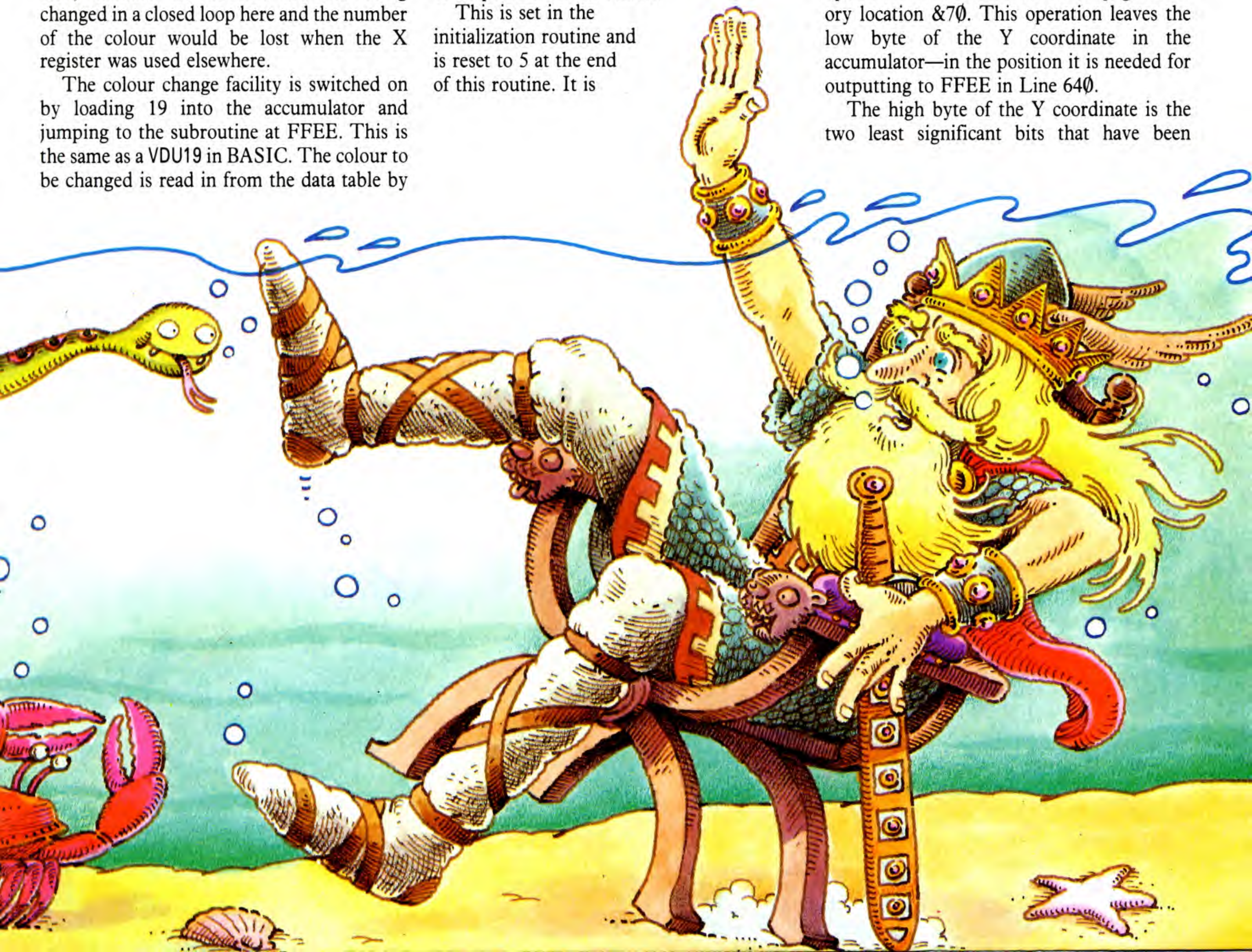
The sea delay is decremented by the instruction in Line &77. If it has counted down to zero, the BEQ instruction branches over the RTS instruction and the processor continues with the routine. Otherwise it returns.

If it is time for the tide to advance, A is loaded with 25 and FFEE is called. This is the same as a BASIC MOVE command. A 4 is then output to FFEE, which gives a MOVE or PLOT4. Ø is then output to FFEE twice, which sets the low byte and the high byte of the X coordinate.

Memory location &88 contains the Y coordinate of the next line of sea up the screen, divided by four. By dividing the Y coordinate by four it can be stored in one byte.

To multiply the contents of &88 by four, they are loaded into the accumulator and two arithmetic shifts to the left are performed. And any bits pushed out of the register by this operation are rotated into the zero-page memory location &7Ø. This operation leaves the low byte of the Y coordinate in the accumulator—in the position it is needed for outputting to FFEE in Line 64Ø.

The high byte of the Y coordinate is the two least significant bits that have been



shifted into &7Ø. But the contents of the rest of the bits are not required. So the contents are loaded into the accumulator and ANDed with 3. This preserves the two least significant bits and sets the rest to zero. And as the result is left in A, it can be output to FFEE simply by jumping to that subroutine with the instruction in Line 67Ø. The cursor is now in position at the left-hand end of the new line of sea about to be printed.

HIGH SEAS

The colour of the new line of sea then has to be set. 18 is loaded into the accumulator and FFEE is called. This gives you a GCOL command. The Ø then output to FFEE makes it give the colour specified directly.

The second parameter output to FFEE is now the colour to be used. And the colour number to be output is picked up from the colour data table by the instruction in Line 73Ø. So the colour to be used is cyan, because the colour picked up from the data table is the last one to have been redefined.

X is then set back to Ø as it is going to be used as a counter to count along the new line of sea. The instruction in Line 77Ø loads up the appropriate sea character with its surf from the sea data table and FFEE is called to print it on the screen. X is incremented to move onto the next character of data in the table. And it is compared to 21, to see whether the whole of the line of sea has been printed.

If it hasn't, the BNE instruction in Line 81Ø branches back to deal with the next character square of the sea. If it has finished, the processor moves on to increment the Y coordinate in &88. 5 is then loaded into the accumulator and stored in &77 to reset the sea delay, and the processor returns.

The following routine turns on the flood tide.

```
        ORG 19678
SEA     LDU #18206
        LDA 18246
        BITA #2
        BEQ SPT
        LDU #18222
SPT     LDX 18247
        LDA #16
SPTI    PSHS A,U
        JSR CHARPR
        PULS U,A
        DECA
        BNE SPTI
        DEC 18246
        BNE SRT
        LDA #10
        STA 18246
        LDX 18247
        LEAX -256,X
        STX 18247
SRT     RTS
CHARPR  EQU 19402
```

To test this program you need to LOAD in the rest of Cliffhanger and RUN the following program:

```
5 POKE &H467F,&H4C:POKE&H4C80,&HF3
10 EXEC19426
20 FORG=1TO160
30 EXEC19678
40 FORH=1TO100:NEXTH,G
50 GOTO50
```

Once this sets the sea going, the tide will rise until the whole screen is filled with water. This will never happen when the game is played though. Willie will have drowned by then and the game will reinitialize.

LITTLE DROPS OF WATER

Although there appears to be a lot of sea, there are, in fact only two characters' worth in the data table. The first occupies the eight locations from 18,2Ø6 onwards and the second occupies the eight from 18,222 onwards. When these two sea characters are printed next to each other over and over in alternating lines, you rapidly build up an ocean.

As the sea is going to be printed on the screen a character at a time, the CHARPR routine is going to be used again. So the relevant parameters have to be loaded into the correct registers. As always, U carries the pointer to the first byte of data, so the data itself acts as the user stack. And X carries the screen position the data is to be printed in. So

U is loaded up with the address of the first byte of the first sea character.

The variable in 18,246 is the so-called sea delay. This controls the movement of the sea. Bit two is used as a flag to tell the processor which sea character was used for the last line.

The contents of the sea delay are loaded into the accumulator and the instruction BITA #2 isolates that particular bit. If it is not set, the BEQ instruction which follows it jumps the processor over the next instruction. But if it is set, the jump is not made and U is loaded with the address of the beginning of the second sea character.

SEE SEA

X is loaded with the contents of memory location 18,247. This location carries the position of the sea that is about to be printed and it has been initialized by the routine in part seven of Cliffhanger on page 1104 of *INPUT* to the bottom left-hand corner of the screen.

A is loaded with 16. This is going to be used as a counter to count across the 16 sea characters that are going to be printed.

This counter needs to be preserved intact while the processor goes off to perform the CHARPR routine, so A is pushed onto the stack. The data for the sea character is going to be used over again, too, because each line of the sea is made up of the same sea character—so U is pushed onto the stack as well. The CHARPR routine is then called and the eight bytes of the appropriate sea character are printed up on the correct place on the screen. CHARPR automatically updates the X register so that it is ready to print the next character alongside the last on the screen.

The data pointer is moved back to the beginning of the appropriate sea character—it has been incremented during the CHARPR routine—by pulling it off the stack. And the A counter is pulled off the stack again too.

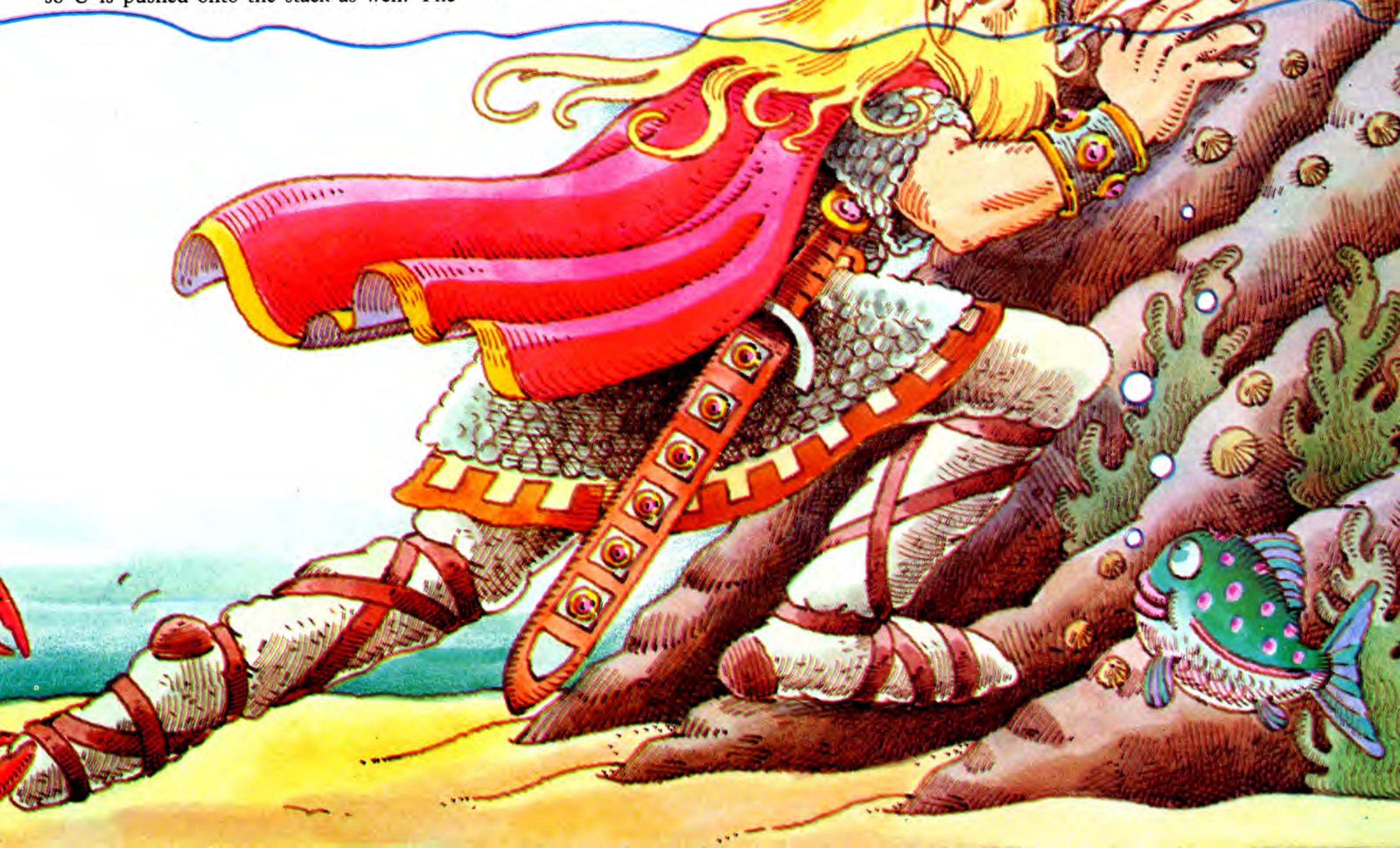
The counter is then decremented and if it hasn't counted down to Ø, the BNE instruction loops back so that the processor prints the sea character in the next screen position along that particular line.

When the counter in A has counted down to Ø, the processor drops out of the loop and proceeds with the next instruction.

SEA SAW

The sea delay is then decremented. If it has not counted down to zero, the BNE instruction jumps the processor over the next few instructions to the end of the routine where it returns.

If it has counted down to zero, the sea delay is reset to 1Ø. The position pointer is loaded back into X, and 256 is subtracted from it. The result is stored back in the sea pointers location, 18,247. So next time this routine is called, the sea is moved up one row.



SQUEEZING OUT A TUNE

Increase the musical power of your computer by reducing the data needed for your tunes. You can use the technique to compress other types of data too

A piece of music, or even a simple tune, played on your computer can be an exhilarating experience—particularly if you have composed and programmed the tune yourself. Naturally, there are difficulties that you must overcome, not the least of which is the large mass of data—perhaps two or three screens full—required to program a typical tune. Besides being tedious to type in, this occupies a lot of memory. This article shows some simple data compression techniques that let you store tunes within BASIC programs, without taking up large amounts of user RAM.

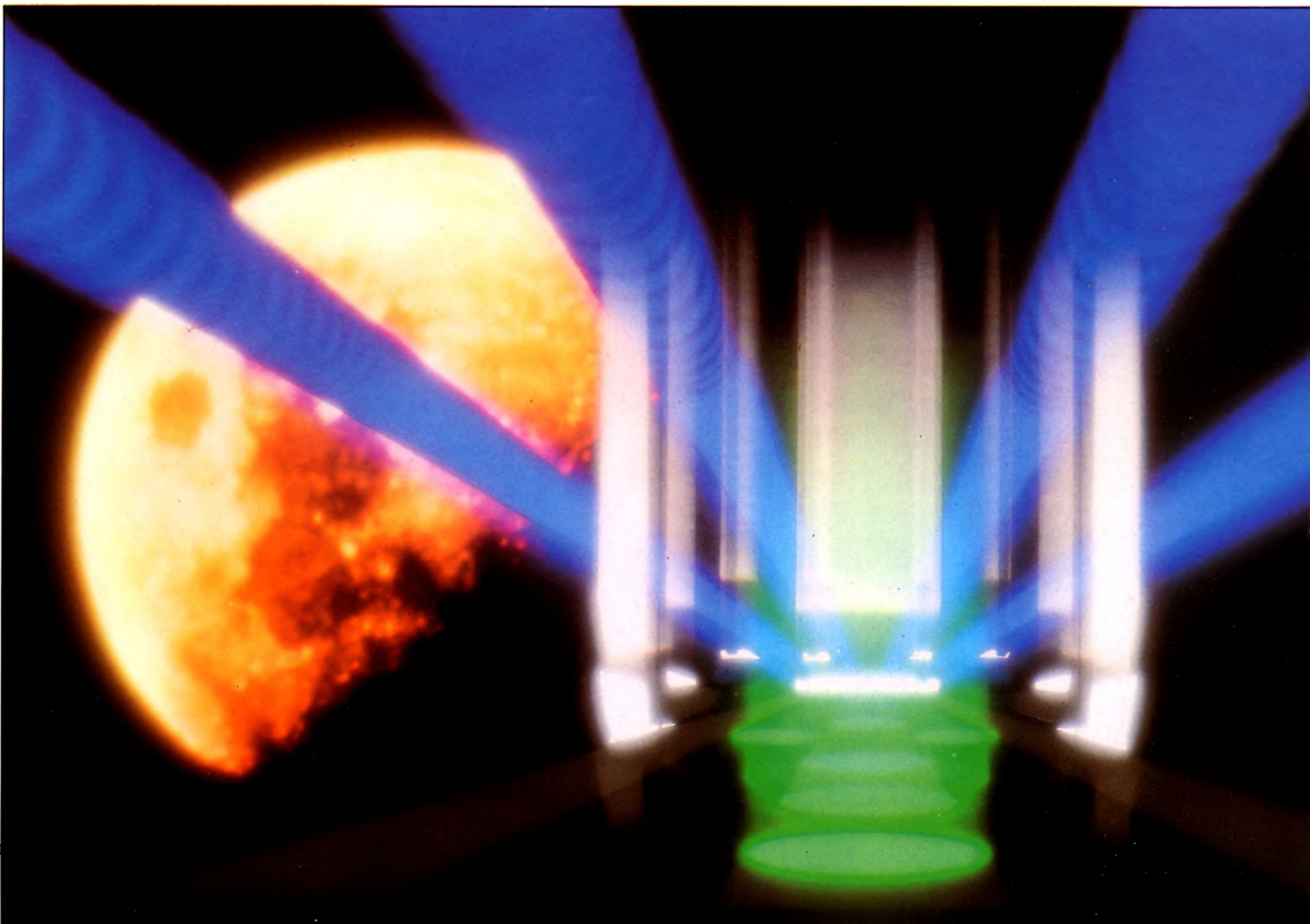
Of course, the need to squeeze the maximum amount of data into the smallest amount of space isn't limited to the generation of tunes. The techniques described in this article can be used for compressing data used in other applications—provided the data is either repetitive or uses only a restricted range of values.

SINGING THE BLUES

Whatever the musical style, most tunes have a similar structure which lends itself to data compression. Suppose you wanted to play a simple 12-bar blues tune, for instance. You would probably write a program in which the pitch values are stored sequentially within DATA statements. Enter and RUN the first program to hear such a tune:

```
10 LET T=.2
20 RESTORE 100
30 READ D
50 IF D=255 THEN GOTO 20
60 BEEP T,D
70 GOTO 30
100 DATA 12,12,15,16,19,19,21,19
110 DATA 12,24,22,21,19,17,16,14
120 DATA 12,12,15,16,19,19,21,19
130 DATA 12,24,22,21,19,17,16,14
140 DATA 17,17,20,21,24,24,26,24
150 DATA 17,24,22,21,19,17,16,14
160 DATA 12,12,15,16,19,19,21,19
170 DATA 12,24,22,21,19,17,16,14
180 DATA 19,19,23,24,26,26,24,23
190 DATA 17,17,20,21,24,24,20,21
200 DATA 12,12,15,16,19,19,21,19
210 DATA 12,24,22,21,19,17,16,14
220 DATA 255
```

```
10 S=54272:FORZ=STOS+24:POKEZ,0:
   NEXTZ:T=100
```



- COMPRESSING A TUNE
- PROGRAMMING A 12-BAR BLUES
- SPLITTING A TUNE INTO SHORT SECTIONS
- FINDING REPETETIVE MINI TUNES
- USING LESS NOTES
- ALTERING THE TEMPO
- PLAYING LONG NOTES

```
20 POKES+5,Ø:POKES+6,24Ø:POKE
   S+24,15:RESTORE
3Ø READK,KK:IFK=-1THEN2Ø
45 POKES+4,33:POKES+11,129
5Ø POKES+1,K:POKES,KK
6Ø FORZ=1TOT:NEXTZ
7Ø POKES+4,32:GOTO3Ø
1ØØ DATA 8,97,8,97,9,247,1Ø,143,12,143,12,
   143,14,24,12,143
11Ø DATA 8,97,16,195,14,239,14,24,12,143,
   11,48,1Ø,143,9,1Ø4
12Ø DATA 8,97,8,97,9,247,1Ø,143,12,143,12,
   143,14,24,12,143
13Ø DATA 8,97,16,195,14,239,14,24,12,143,
   11,48,1Ø,143,9,1Ø4
14Ø DATA 11,48,11,48,13,78,14,24,16,195,16,
   195,18,2Ø9,16,195
15Ø DATA 11,48,16,195,14,239,14,24,12,143,
   11,48,1Ø,143,9,1Ø4
16Ø DATA 8,97,8,97,9,247,1Ø,143,12,143,12,
   143,14,24,12,143
17Ø DATA 8,97,16,195,14,239,14,24,12,143,
   11,48,1Ø,143,9,1Ø4
18Ø DATA 12,143,12,143,15,21Ø,16,195,18,
   2Ø9,18,2Ø9,16,195,15,21Ø
19Ø DATA 11,48,11,48,13,78,14,24,16,195,16,
   195,13,78,14,24
2ØØ DATA 8,97,8,97,9,247,1Ø,143,12,143,12,
   143,14,24,12,143
21Ø DATA 8,97,16,195,14,239,14,24,12,143,
   11,48,1Ø,143,9,1Ø4
999 DATA -1,Ø
```

```
1Ø S=36874:FORZ=STOS+4:POKEZ,Ø:
   NEXTZ:T=2ØØ
2Ø POKES+4,15:RESTORE
3Ø READK:IFK=-1THEN2Ø
5Ø FOR Z=ØTO2:POKES+Z,K:NEXTZ
6Ø FORZ=1TOTNEXTZ
7Ø FORZ=ØTO2:POKES+Z,Ø:NEXTZ:GOTO
   3Ø
1ØØ DATA 173,173,185,189,2ØØ,2ØØ,2Ø6,2ØØ
11Ø DATA 173,214,2Ø8,2Ø6,2ØØ,192,189,181
12Ø DATA 173,173,185,189,2ØØ,2ØØ,2Ø6,2ØØ
13Ø DATA 173,214,2Ø8,2Ø6,2ØØ,192,189,181
14Ø DATA 192,192,2Ø3,2Ø6,214,214,218,214
15Ø DATA 192,214,2Ø8,2Ø6,2ØØ,192,189,181
16Ø DATA 173,173,185,189,2ØØ,2ØØ,2Ø6,2ØØ
17Ø DATA 173,214,2Ø8,2Ø6,2ØØ,192,189,181
18Ø DATA 2ØØ,2ØØ,211,214,218,218,214,211
19Ø DATA 192,192,2Ø3,2Ø6,214,214,2Ø3,2Ø6
2ØØ DATA 173,173,185,189,2ØØ,2ØØ,2Ø6,2ØØ
21Ø DATA 173,214,2Ø8,2Ø6,2ØØ,192,189,181
999 DATA-1
```

```
1Ø T=4
2Ø RESTORE1ØØ
3Ø READD
5Ø IFD=255THEN2Ø
6Ø SOUND1,-15,D,T:SOUND1,Ø,Ø,1
7Ø GOTO3Ø
1ØØ DATA 1ØØ,1ØØ,112,116,128,128,136,128
11Ø DATA 1ØØ,148,14Ø,136,128,12Ø,116,1Ø8
12Ø DATA 1ØØ,1ØØ,112,116,128,128,136,128
13Ø DATA 1ØØ,148,14Ø,136,128,12Ø,116,1Ø8
14Ø DATA 12Ø,12Ø,132,136,148,148,156,148
15Ø DATA 12Ø,148,14Ø,136,128,12Ø,116,1Ø8
16Ø DATA 1ØØ,1ØØ,112,116,128,128,136,128
17Ø DATA 1ØØ,148,14Ø,136,128,12Ø,116,1Ø8
18Ø DATA 128,128,144,148,156,156,148,144
19Ø DATA 12Ø,12Ø,132,136,148,148,132,136
2ØØ DATA 1ØØ,1ØØ,112,116,128,128,136,128
21Ø DATA 1ØØ,148,14Ø,136,128,12Ø,116,1Ø8
22Ø DATA 255
```

```
1Ø T=3
2Ø RESTORE
3Ø READD
5Ø IFD=255THEN2Ø
6Ø SOUNDD,T
7Ø GOTO 3Ø
1ØØ DATA 175,175,189,193,2Ø4,2Ø4,21Ø,2Ø4
11Ø DATA 175,218,213,21Ø,2Ø4,197,193,185
12Ø DATA 175,175,189,193,2Ø4,2Ø4,21Ø,2Ø4
13Ø DATA 175,218,213,21Ø,2Ø4,197,193,185
14Ø DATA 197,197,2Ø7,21Ø,218,218,223,218
15Ø DATA 197,218,213,21Ø,2Ø4,197,193,185
16Ø DATA 175,175,189,193,2Ø4,2Ø4,21Ø,2Ø4
17Ø DATA 175,218,213,21Ø,2Ø4,197,193,185
18Ø DATA 2Ø4,2Ø4,216,218,223,223,218,216
19Ø DATA 197,197,2Ø7,21Ø,218,218,2Ø7,21Ø
2ØØ DATA 175,175,189,193,2Ø4,2Ø4,21Ø,2Ø4
21Ø DATA 175,218,213,21Ø,2Ø4,197,193,185
22Ø DATA 255
```

The data for the Commodore is twice as long as for other micros, because each note is specified by two pitch values—one for low byte and another for high byte.

The variable T sets a time factor to control the speed of the tune. Line 2Ø sets the data pointer to the first line of data, then the program loops between Lines 3Ø and 7Ø reading the pitch values—each of the numbers in the DATA statements—in turn into the sound statement at Line 6Ø (4Ø to 7Ø on the Commodores). Notice that, on the Acorns, there is a second sound statement which plays a note of zero loudness and zero pitch for a twentieth of a second. This 'dummy note' is a period of silence to separate the true notes. Line 5Ø detects the end of the tune, which is marked by the arbitary value 255; on the Commodores Line 3Ø detects value -1 instead.

If you wanted to program a rest at some point in the tune, you could insert another arbitary value (254, say) and include a test at

Line 4Ø to detect it. If the test was successful the program would branch to a line that set a delay, then returned control to Line 3Ø to continue the tune.

As the program stands, it has the effect of passing pitch values to the sound-handling section of the micro, where they are executed sequentially—as they appear in the data.

Although this program works quite adequately you can see from the listings that, even for this short tune, a large amount of data is required. This is tedious to type in, and takes up more than its fair share of memory. It brings another drawback too: while the data statements are being processed (that is, while the tune is being played) your micro can't get on with any other task.

Some micros, such as the Acorns, partly solve this problem by having a sound buffer which can hold data for up to six sound statements. If the buffer has room for all the data used in a particular tune then the computer is free to continue with any other processing—but even here, while there is sound to be processed, the micro must attend to this task. And a sound buffer does nothing to relieve the tedium of entering the data, or to reduce the amount of memory needed.

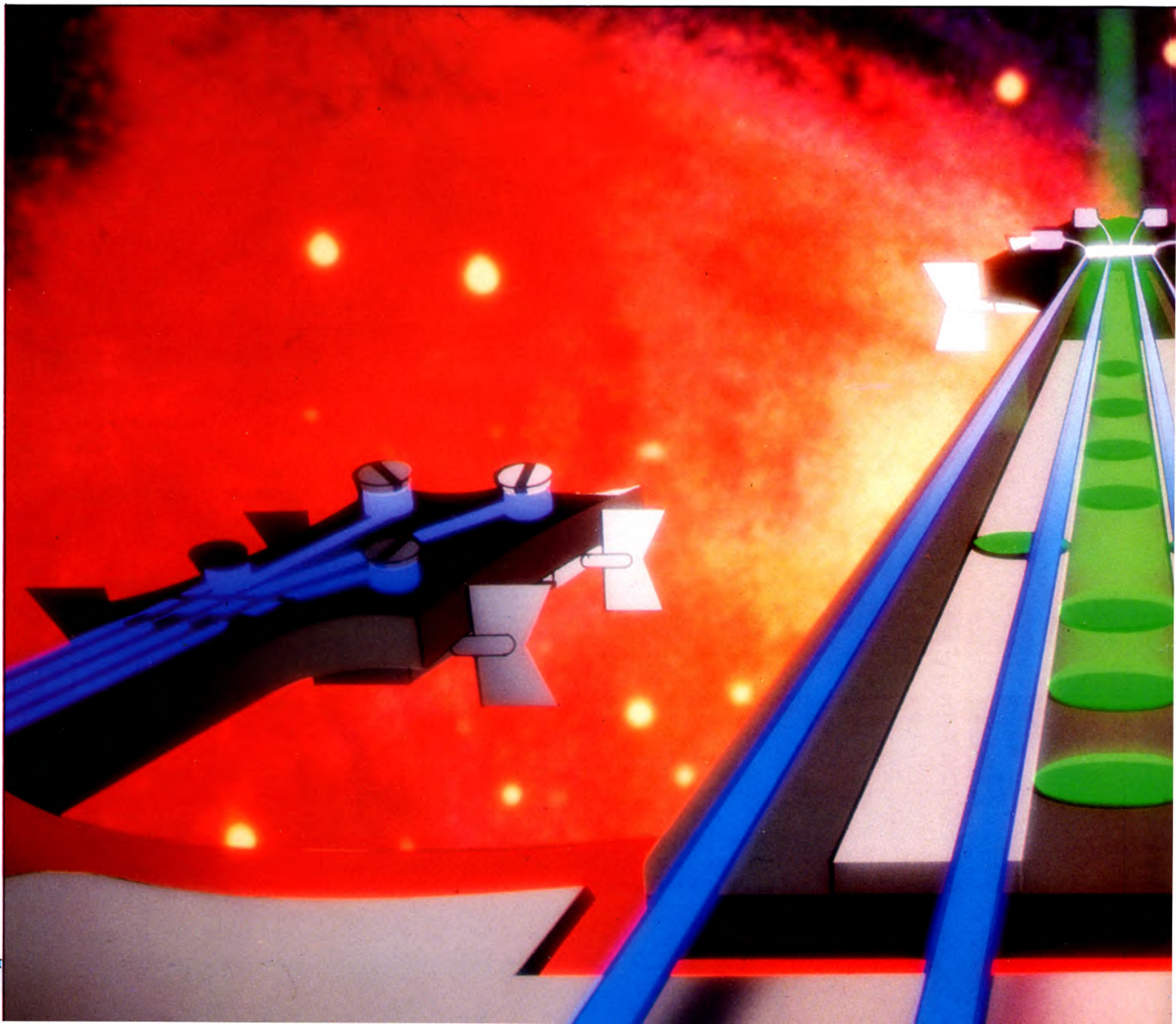
So what's really needed, apart from the obvious solution of writing very short tunes, is to find some way of compacting or *compressing* the data to take up as little space, and make it quick to enter and process.

GETTING THE TREND

Data compression relies on your data having some underlying trend or trends. The more of these trends that you can recognize, the greater can be the degree of compression.

The first step in analysing trends within the data for a tune is to actually play or listen to the tune on an instrument, and try to identify passages that sound alike. Write the tune down on paper, ignoring staves, time signatures and other musical conventions, and concentrating on the pitch of each note.

It is a simple matter to write the letter of each note sequentially—as they occur within the tune—in a straight line. That is fine if each note lasts for the same length of time—one beat, say. But what happens if any note lasts for more than one beat? The program is much simpler if we assume that each note lasts for the same period—that is, if T is given a constant value. If the value of T is allowed to change for each note, then along with the pitch you would have to store the duration of that note, doubling the amount of data. To take into account notes that do last for more than one beat, you can simply enter the same



note more than once—for example, for three beats, you would enter the same pitch value three times.

When you have written down the blues tune, it should look like the values in Table 1 below.

Table 1

G_1	G_1	$A\#_1$	B_1	D_1	D_1	E_1	D_1	G_1
G_2	F_1	E_1	D_1	C_1	B_1	A_1		
G_1	G_1	$A\#_1$	B_1	D_1	D_1	E_1	D_1	G_1
G_2	F_1	E_1	D_1	C_1	B_1	A_1		
C_1	C_1	$D\#_1$	E_1	G_2	G_2	A_2	G_2	C_1
G_2	F_1	E_1	D_1	C_1	B_1	A_1		
G_1	G_1	$A\#_1$	B_1	D_1	D_1	E_1	D_1	G_1
G_2	F_1	E_1	D_1	C_1	B_1	A_1		
D_1	D_1	$F\#_1$	G_2	A_2	A_2	G_2	$F\#_1$	
C_1	C_1	$D\#_1$	E_1	G_2	G_2	$D\#_1$	E_1	
G_1	G_1	$A\#_1$	B_1	D_1	D_1	E_1	D_1	G_1
G_2	F_1	E_1	D_1	C_1	B_1	A_1		

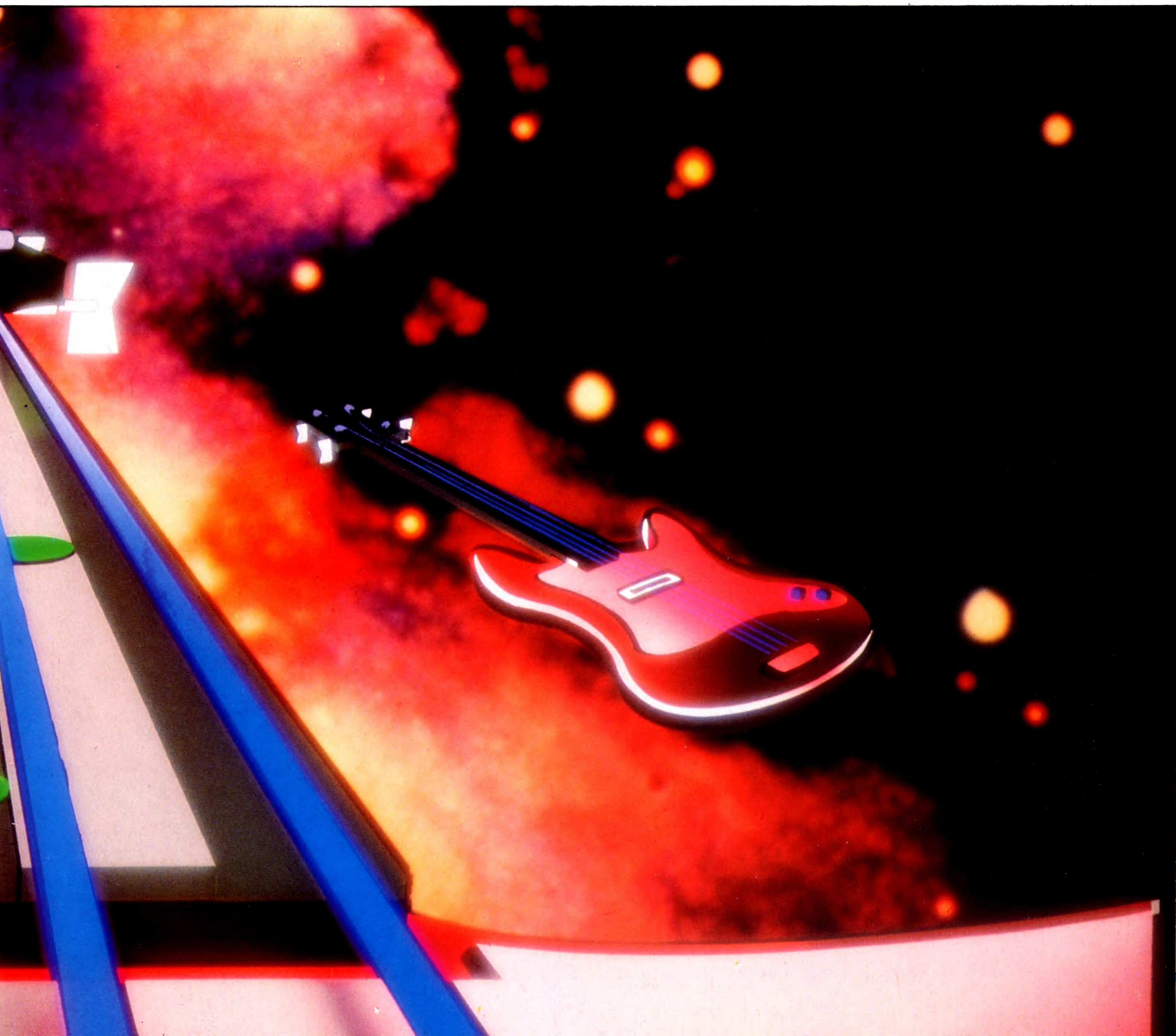
If you study the Table, you'll be able to see that the whole tune is made up from just five different series of notes, or 'mini' tunes, most of them repeated several times as shown in the next Table, below.

Table 2

$T_1=G_1$	G_1	$A\#_1$	B_1	D_1	D_1	E_1	D_1	G_1
$T_2=G_2$	F_1	E_1	D_1	C_1	B_1	A_1		
$T_3=C_1$	C_1	$D\#_1$	E_1	G_2	G_2	A_2	G_2	C_1
$T_4=D_1$	D_1	$F\#_1$	G_2	A_2	A_2	G_2	$F\#_1$	
$T_5=C_1$	C_1	$D\#_1$	E_1	G_2	G_2	$D\#_1$	E_1	

Now you have a method for data compression—instead of entering all the notes of each of the mini tunes every time they occur you enter the notes of each mini tune only once, together with a short series of codes describing the sequence in which the mini tunes are to be played. The trade-off in this saving of memory is that the program becomes longer, as it has to work out which data to process at any one time. The second program shows this:

```
10 LET C=Ø: LET T=.2
20 RESTORE 1ØØ
30 FOR N=1 TO C+1: READ P:
   NEXT N
40 IF P=Ø THEN GOTO 1Ø
50 RESTORE P
60 READ N
70 IF N>=255 THEN LET C=C+1:
   GOTO 2Ø
80 BEEP T,N
90 GOTO 6Ø
100 DATA 11Ø,12Ø,11Ø,12Ø,13Ø,12Ø,11Ø,
    12Ø,14Ø,15Ø,11Ø,12Ø,Ø
110 DATA 12,12,15,16,19,19,21,19,12,255
```



```
120 DATA 24,22,21,19,17,16,14,255
130 DATA 17,17,2Ø,21,24,24,26,24,17,255
140 DATA 19,19,23,24,26,26,24,23,255
150 DATA 17,17,2Ø,21,24,24,2Ø,21,255
```

```
1 S=54272:FORZ=STOS+24:POKEZ,Ø:
    NEXTZ
2 POKES+5,Ø:POKES+6,24Ø:POKES
    +24,15
1Ø C=Ø:T=1ØØ
2Ø RESTORE
27 FORZ=1TOC+1:READP:NEXTZ
28 IFP=ØTHEN1Ø
29 RESTORE:FORW=1TOP:READWW:
    NEXTW
3Ø READK,KK:IFK=-1THENC=C+1:
    GOTO2Ø
4Ø POKES+4,33
5Ø POKES+1,K:POKES,KK
6Ø FORZ=1TOT:NEXTZ
7Ø POKES+4,32:GOTO3Ø
1ØØ DATA 13,33,13,33,49,33,13,33,69,87,13,
    33,Ø
11Ø DATA 8,97,8,97,9,247,1Ø,143,12,143,12,
    143,14,24,12,143,8,97,-1,Ø
12Ø DATA 16,195,14,239,14,24,12,143,11,48,
    1Ø,143,9,1Ø4,-1,Ø
13Ø DATA 11,48,11,48,13,78,14,24,16,195,16,
    195,18,2Ø9,16,195,11,48,-1,Ø
14Ø DATA 12,143,12,143,15,21Ø,16,195,18,
    2Ø9,18,2Ø9,16,195,15,21Ø,-1,Ø
15Ø DATA 11,48,11,48,13,78,14,24,16,195,16,
    195,13,78,14,24,-1,Ø
```

```
1 S=36874:FORZ=STOS+4:POKEZ,Ø:
    NEXTZ
2 POKES+4,15
1Ø C=Ø:T=2ØØ
2Ø RESTORE
27 FORZ=1TOC+1:READP:NEXTZ
28 IFP=ØTHEN1Ø
29 RESTORE:FORW=1TOP:READWW:
    NEXTW
3Ø READK:IFK=-1THENC=C+1:
    GOTO2Ø
5Ø POKES+2,K
6Ø FORZ=1TOT:NEXTZ
7Ø POKES+2,Ø:GOTO3Ø
1ØØ DATA 13,23,13,23,31,23,13,23,41,5Ø,13,
    23,Ø
11Ø DATA 173,173,185,189,2ØØ,2ØØ,2Ø6,2ØØ,
    173,-1
12Ø DATA 214,2Ø8,2Ø6,2ØØ,192,189,181,-1
13Ø DATA 192,192,2Ø3,2Ø6,214,214,218,214,
    192,-1
14Ø DATA 2ØØ,2ØØ,211,214,218,218,214,211,
    -1
15Ø DATA 192,192,2Ø3,2Ø6,214,214,2Ø3,2Ø6,
    -1
```

```
1Ø C=Ø:T=4
2Ø RESTORE 1ØØ
3Ø FOR N=1 TO C+1:READ P:NEXT
4Ø IF P=Ø THEN 1Ø
5Ø RESTORE (1ØØ+P)
6Ø READ N
7Ø IF N=255 THEN C=C+1:GOTO 2Ø
8Ø SOUND1,-15,N,T:SOUND1,Ø,Ø,1
9Ø GOTO 6Ø
1ØØ DATA 1Ø,2Ø,1Ø,2Ø,3Ø,2Ø,1Ø,2Ø,4Ø,5Ø,
    1Ø,2Ø,Ø
11Ø DATA 1ØØ,1ØØ,112,116,128,128,136,128,
    1ØØ,255
12Ø DATA 148,14Ø,136,128,12Ø,116,1Ø8,255
13Ø DATA 12Ø,12Ø,132,136,148,148,156,148,
    12Ø,255
14Ø DATA 128,128,144,148,156,156,148,144,
    255
15Ø DATA 12Ø,12Ø,132,136,148,148,132,136,
    255
```

```
1 DIMA(5,1):FORK=1TO13:READP:NEXT:
    GOTO3
2 READ P:IF P<>255 THEN2
3 N=N+1:A(N,Ø)=PEEK(51):A(N,1)=PEEK
    (52):IF N<5 THEN2
1Ø C=Ø:T=3
2Ø RESTORE
3Ø FORN=1 TO C+1:READ P:NEXT
4Ø IF P=Ø THEN 1Ø
5Ø POKE51,A(P,Ø):POKE52,A(P,1)
6Ø READ N
7Ø IF N=255 THEN C=C+1:GOTO2Ø
8Ø SOUND N,T
9Ø GOTO6Ø
1ØØ DATA 1,2,1,2,3,2,1,2,4,5,1,2,Ø
11Ø DATA 175,175,189,193,2Ø4,2Ø4,21Ø,2Ø4,
    175,255
12Ø DATA 218,213,21Ø,2Ø4,197,193,185,255
13Ø DATA 197,197,2Ø7,21Ø,218,218,223,218,
    197,255
14Ø DATA 2Ø4,2Ø4,216,218,223,223,218,216,
    255
15Ø DATA 197,197,2Ø7,21Ø,218,218,2Ø7,21Ø,
    255
```

Notice that the amount of data required to play the blues tune is much reduced—on the Spectrum, for example, from 97 bytes to 59 bytes. RUN the program and verify that the tune is the same as that played by the first program. Spectrum users will note that the timing of the tune is a little odd, and more will be said about this later.

The data for the mini tunes is at Lines 11Ø to 15Ø, and the data for the master sequence—the order in which the mini tunes are played—is at Line 1ØØ. The loop at Line 3Ø sets P within the master sequence to select which mini tune is to be played. In fact, the master sequence is a list of line numbers (or numbers that combine with an offset to give line numbers) where the data for mini tunes is listed.

Once the mini tune to be played is calculated, Line 5Ø sets the data pointer to the start of the appropriate line of data. The Dragon, Tandy and Commodores do not allow RESTORE to a line within a block of data, but only to the first line of data. So to point to the data for a particular mini tune, the Commodores use FOR ... NEXT loops (Lines 27 and 29). On the Dragon and Tandy, the data points are recorded at Line 3, then recalled at Line 5Ø to play each mini tune.

Line 1Ø sets C to count the number of mini tunes that have been played. The duration of a single note is set by T (except on the Commodores) which controls the overall tempo of the tune. Line 4Ø checks for the last

Q+A

On my Acorn computer, if I enter the same pitch value more than once in a row I get two distinct notes, rather than a single note of double duration. How can I cure this?

The problem is caused by the dummy note which is inserted to prevent successive notes being played too quickly and merging into one indistinct sound. The answer is to use a pitch value of 256 *greater* than the actual value of the note you want played. By inserting a test to check for values greater than 256 (as was mentioned for values of 254 and 255) you can branch the program to a routine that removes the dummy note and allows the sounds to merge into one, longer note.

Why use a value exactly 256 greater than the note you want? The Acorns recognize values over 256 as illegal—but instead of issuing an error message, they subtract 256 from the value repeatedly until the remainder is less than 256, then play the note associated with this number. For example, if you want to play a note of pitch 1ØØ enter it as pitch 356—the Acorn will subtract 256 from the entered number and play a note of pitch 1ØØ, which is what you really wanted.

mini tune, which is marked by Ø at Line 1ØØ. The end of each mini tune is marked by 255. After each has been played, the program loops back to read the master sequence again to get the line number of the next tune.

There are many ways of splitting a tune into a master sequence and mini tunes. Generally, the shorter the mini tunes, the longer will be the master sequence. You should aim for a balance in which the mini tunes are small, but not so small that the advantage of using the system is lost by increasing the size of the master sequence too much.

DIVIDE AND CONQUER

The second data compression technique is even more efficient. It relies on the fact that although a large number of notes is available on some micros—typically 256—only a few of these are used in any one tune. And it is wasteful to store these using a system that is designed to store many more different values.

Each memory location of your eight-bit micro can store a decimal number in the range Ø to 255. If you can restrict the range of numbers you want to store, then you may be able to pack two of them—one into four bits—into each location. For example, the eight-bit binary number 1Ø1ØØØ1Ø can be thought of as either the single decimal number 162, or the two decimal numbers 1Ø (from the leftmost four bits 1Ø1Ø) and 2 (from the rightmost four bits ØØ1Ø).

Halving the number of bits available for storing each number restricts the range of decimal numbers you can store quite drastically—to between Ø and 15. But this may be sufficient—the number of different notes in a simple tune often doesn't exceed 16.

To make use of this arrangement for data compression, you should restrict the number of different notes you store to 15, leaving the sixteenth combination as a control code. As before, shortening the data generally increases the amount of program you need to process it. In this case, the program has to work out which pitch value to use for each of the abbreviated, coded forms it finds in the DATA statements. And *you* have to do quite a lot of work in preparing the data for the program by working out how many different notes are used in the tune, then arranging them in order of ascending pitch starting with the lowest note of the first octave—in this case G_1. Now work out what the coded forms of the data should be. For the tune you have played, there are 12 notes: G_1, A_1, $A\#_1$, B_1, C_1, D_1, $D\#_1$, E_1, F_1, $F\#_1$, G_2 and A_2. Enter and RUN the third program to see how the listing progresses:

```
12 GOSUB 1ØØØ: LET T=.15: LET NT=13Ø:
   LET MS=17Ø: LET MT=21Ø:GOSUB 3ØØ
9Ø STOP
1ØØ DATA 12,14,15,16,17,19,2Ø,21,22,23,24,
   26,Ø,Ø,Ø,Ø
2ØØ DATA 1,2,1,2,3,2,1,2,4,5,1,2,Ø
21Ø DATA Ø,35,85,117,15,255
22Ø DATA 168,117,67,31,255
23Ø DATA 68,1Ø3,17Ø,186,79,255
24Ø DATA 85,154,187,169,255
25Ø DATA 68,1Ø3,17Ø,1Ø3,255
31Ø RESTORE NT
32Ø FOR N=2341Ø TO 23425
33Ø READ X: POKE N,X: NEXT N
345 LET NM=Ø
35Ø RESTORE MS: LET HL=23426
36Ø READ X
365 IF X>NM THEN LET NM=X
37Ø IF X=Ø THEN GOTO 4ØØ
38Ø POKE HL,X: LET HL=HL+1:
   GOTO 36Ø
4ØØ POKE HL,X: LET HL=HL+1
4Ø1 LET X=HL: GOSUB 6ØØ
4Ø2 POKE 234Ø3,LSB
4Ø3 POKE 234Ø4,MSB
43Ø RESTORE MT
44Ø FOR N=1 TO NM
45Ø READ X
46Ø IF X=255 THEN GOTO 5ØØ
47Ø POKE HL,X: LET HL=HL+1:
   GOTO 45Ø
5ØØ POKE HL,X: LET HL=HL+1
51Ø NEXT N
511 RANDOMIZE USR 23371
512 POKE 234Ø9,Ø
53Ø LET X=USR 23296
54Ø IF X=255 THEN RETURN
55Ø BEEP T,X: GOTO 53Ø
6ØØ LET MSB=INT (X/256)
61Ø LET LSB=X-(MSB*256): RETURN
1ØØØ RESTORE 2ØØØ: LET TO=Ø: LET
   L=2ØØØ
1Ø3Ø FOR N=23296 TO 23296+111
   STEP 8
1Ø4Ø FOR K=Ø TO 7: READ A: LET
   TO=TO+A: POKE K+N,A: NEXT K
1Ø5Ø READ A: IF A<>TO THEN
   GOTO 1Ø8Ø
1Ø6Ø LET L=L+1Ø: LET TO=Ø: NEXT N
1Ø65 RESTORE : RETURN
1Ø8Ø PRINT "DATA ERROR AT LINE ";L:
   STOP
2ØØØ DATA 42,1Ø9,91,235,42,111,91,58,779
2Ø1Ø DATA 113,91,7Ø,183,2Ø2,24,91,175,949
2Ø2Ø DATA 8,35,62,15,16Ø,195,36,91,6Ø2
2Ø3Ø DATA 61,1,8,2Ø3,56,2Ø3,56,2Ø3,791
2Ø4Ø DATA 56,2Ø3,56,12Ø,254,15,2Ø2,65,971
2Ø5Ø DATA 91,34,111,91,235,34,1Ø9,91,796
2Ø6Ø DATA 8,5Ø,113,91,33,114,91,22,522
2Ø7Ø DATA Ø,8,95,25,126,6,Ø,79,339
2Ø8Ø DATA 2Ø1,26,19,183,194,81,91,1,796
2Ø9Ø DATA 255,Ø,2Ø1,17,13Ø,91,195,65,954
21ØØ DATA 91,71,42,1Ø7,91,43,62,255,762
211Ø DATA 16,5,35,175,195,1Ø,91,35,562
212Ø DATA 19Ø,194,1Ø3,91,195,88,91,35,987
213Ø DATA 195,96,91,Ø,Ø,Ø,Ø,Ø,382
```

Microtip

More than one tune

As an exercise, try to code additional tunes for the last program. At first, aim to add about three tunes. Work out the master sequences and mini tune components, then store them in data statements. You will need to reinitialize the array variables—X()— for the master note table of each tune before it is to be played. Some renumbering might be necessary.

```
1Ø S=54272:FORZ=STOS+24:POKEZ,Ø:
   NEXTZ
2Ø POKES+5,Ø:POKES+6,24Ø:POKE
   S+24,15
23 DIMX(16),XX(16):RESTORE:FORN=1TO
   16:READX(N),XX(N):NEXTN
25 C=Ø:T=1ØØ
26 RESTORE:FORW=1TO32:READWW:NEXT
   W
27 FORZ=1TOC+1:READP:NEXTZ
28 IFP=ØTHEN25
29 RESTORE:FORW=1TOP+4Ø:READWW:
   NEXTW
5Ø READN:SS=N
6Ø N=INT(N/16)
7Ø IFN=15THENC=C+1:GOTO 26
8Ø GOSUB 13Ø
9Ø N=SS:N=15 AND N
1ØØ IF N=15 THEN C=C+1:GOTO 26
11Ø GOSUB 13Ø
12Ø GOTO 5Ø
13Ø POKES+4,33
14Ø POKES+1,X(N+1):POKES,XX(N+1)
15Ø FORZ=1TOT:NEXTZ
16Ø POKES+4,32:RETURN
45Ø DATA 8,97,9,1Ø4,9,247,1Ø,143,11,48,12,
   143,13,78,14,24,14,239,15,21Ø,16,195
46Ø DATA 18,2Ø9,Ø,Ø,Ø,Ø,Ø,Ø,Ø,Ø
1ØØØ DATA 5,1Ø,5,1Ø,15,1Ø
1Ø1Ø DATA 5,1Ø,2Ø,25,5,1Ø,Ø
111Ø DATA Ø,35,85,117,15
112Ø DATA 168,117,67,31,255
```

```
1130 DATA 68,103,170,186,79
1140 DATA 85,154,187,169,255
1150 DATA 68,103,170,103,255
```

```
10 S=36874:FORZ=STOS+4:POKEZ,0:
   NEXTZ
20 POKES+4,15
23 DIMX(16):RESTORE:FORN=1TO16:READ
   X(N):NEXTN
25 C=0:T=200
26 RESTORE:FORW=1TO16:READWW:NEXT
   W
27 FORZ=1TOC+1:READP:NEXTZ
28 IFP=0THEN25
29 RESTORE:FORW=1TOP+24:READWW:
   NEXTW
50 READN:SS=N
60 N=INT(N/16)
70 IF N=15 THEN C=C+1:GOTO 26
80 GOSUB 130
90 N=SS:N=15 AND N
100 IF N=15 THEN C=C+1:GOTO 26
110 GOSUB 130
120 GOTO 50
130 POKES+2,X(N+1)
150 FORZ=1TOT:NEXTZ
160 POKES+2,0:RETURN
450 DATA 173,181,185,189,192,200,203,206,
    208,211,214,218,0,0,0,0
1000 DATA 5,10,5,10,15,10
1010 DATA 5,10,20,25,5,10,0
1110 DATA 0,35,85,117,15
1120 DATA 168,117,67,31,255
1130 DATA 68,103,170,186,79
1140 DATA 85,154,187,169,255
1150 DATA 68,103,170,103,255
```

```
10 DIM X(16):RESTORE450: FOR N=1 TO
   16:READ X(N):NEXT
20 C=0:T=4
30 RESTORE 1000:FOR N=1 TO C+1:READ
   P:NEXT:IF P=0 THEN 20
40 RESTORE (1100+P)
50 READ N:S=N
60 N=N□DIV 16
70 IF N=15 THEN C=C+1:GOTO 30
80 PROCSO
90 N=S□MOD 16
100 IF N=15 THEN C=C+1:GOTO 30
110 PROCSO
120 GOTO 50
130 DEF PROCSO:SOUND1,-15,X(N+1),T:
    SOUND1,0,0,1:ENDPROC
450 DATA 100,108,112,116,120,128,132,136,
    140,144,148,156,0,0,0,0
1000 DATA 10,20,10,20,30,20
1010 DATA 10,20,40,50,10,20,0
1110 DATA 0,35,85,117,15
1120 DATA 168,117,67,31
```

```
1130 DATA 68,103,170,186,79
1140 DATA 85,154,187,169,255
1150 DATA 68,103,170,103,255
```

```
10 DIMA(5,1),X(16):FORK=1TO16:READX
   (K):NEXT:A(0,0)=PEEK(51):A(0,1)=
   PEEK(52)
15 FORK=1TO13:READP:NEXT:GOTO30
20 FORK=1TO5:READP:NEXT
30 N=N+1:A(N,0)=PEEK(51):A(N,1)=
   PEEK(52):IF N<5 THEN20
40 C=0:T=3
50 POKE51,A(0,0):POKE52,A(0,1):
   FORN=1TO C+1:READ P:NEXT:
   IF P=0 THEN 40
60 POKE51,A(P,0):POKE52,A(P,1)
65 READ N:S=N
70 N=INT(N/16)
75 IF N=15 THEN C=C+1:GOTO 50
80 GOSUB 130
90 N=S:N=15ANDN
100 IF N=15 THEN C=C+1:GOTO 50
110 GOSUB 130
120 GOTO 65
130 SOUND X(N+1),T:RETURN
450 DATA 175,185,189,193,197,
    204,207,210,213,216,218,223,0,0,0,0
1000 DATA 1,2,1,2,3,2
1010 DATA 1,2,4,5,1,2,0
1110 DATA 0,35,85,117,15
1120 DATA 168,117,67,31,255
1130 DATA 68,103,170,186,79
1140 DATA 85,154,187,169,255
1150 DATA 68,103,170,103,255
```

The pitch values of these notes are coded in the data statement at Line 450 (100 on the Spectrum). Note that on all except the Commodore 64, there must be 16 items at this line. The Commodore 64 listing has 32 items, because two parameters specify each pitch value. In all cases, only 12 notes are coded, so the remaining four (or eight) spaces are filled out with zeros. The four binary bits (called a nybble) that make up each note of the mini tunes are coded at Lines 1000 and 1010 (200 and 210 on the Spectrum).

To see how these values are calculated, take mini tune T1 as an example. Label each item in the master note sequence at Line 450 (100 on the Spectrum). The first note at this line is 0, the second is 1, the third is 2, and so on to the last note, which is 15. Now write T1 in terms of these labels, so that T1 = 0, 0, 2, 3, 5, 5, 7, 5, 0, 15. Note that the 15 at the end of the list is used as an 'end of mini tune' indicator. Each of these values is stored in one nybble, so combine them in pairs to make up bytes.

The first nybble pair is 0 and 0, which in binary gives 0000 and 0000. When combined, the result is decimal 0. So the first item of data in mini tune T1 at Line 1110 (210 on the Spectrum) is 0. The next two items in T1 are 2 and 3. These are 0010 and 0011, which combine to give 00100011 in binary, or decimal 35. This is the second item of data in T1 at Line 1110 (210 for the Spectrum). Similarly, T2 becomes 10, 9, 7, 5, 4, 3, 1, 15. So 10 and 9 combine as 1010 and 1001 in binary, or decimal 168—the first value for T2 at Line 1110 (220 on the Spectrum). This method is continued until all the data for the mini tunes (Lines 1110 to 1150 or 210 to 250 on the Spectrum) are calculated. As in the previous programs, the master sequence (Lines 1000 and 1010 or 200 for the Spectrum) is made up of values that point to the lines at which the required mini tune data is listed—in the order that they are played.

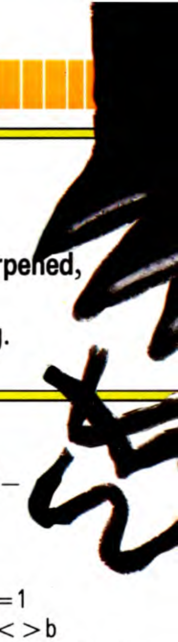
SPOT THAT TUNE

Another important section of the program deals with extracting the encoded nybbles from the one-byte decimal numbers. On the Spectrum, decoding is achieved by storing the byte being processed in a variable (Line 401), then calling a subroutine (Lines 600 and 610), which separates the byte into nybbles. After further processing, these nybbles are used with data from the master sequence to sound a note (Line 550).

On the other micros, decoding is achieved at Lines 50 to 100. Line 50 stores the value of the byte being processed (N). Line 60 extracts the left-hand nybble, then Line 70 checks for the end of a mini tune. Line 80 calls a subroutine to play the note encoded by this nybble, and Line 90 extracts the right-hand nybble, using logical AND. As before, the note is sounded. Notice that the byte being processed is first stored, otherwise it could not yield the second nybble.

When this program is RUN, users of the Spectrum should notice that the disturbing irregularity in the tempo, caused by the compression in the second program, is cured. The defect was due to the extra processing required to run the BASIC program while the data pointers were realigned. The last program reduces this extra processing time by a section of machine code (Lines 1000 to 2130), which is the reason that the program is longer, with different line numbers.

To alter the speed of the tune change the value of the variable T. This is set in Line 12 on the Specrum, Line 25 on the Commoores, Line 20 on the Acorn and Line 40 on the Dragon and Tandy. However you should be aware that the time lag between executing one mini tune and finding the next becomes more marked as the value of T is decreased.



GO OUT WITH A BANG

The game is initialized. Freddy is waiting on the ladder, his arrows are sharpened, the balloons are inflated, and the spider is salivating.

In part one of Freddy and the spider from Mars you entered the initialization and graphics routines. Now lace them together by adding animation routines.

Now add these lines, and you'll have the complete game:

THE MAIN LOOP

```
10 CLEAR 65287
20 GOSUB 1000
30 GOSUB 3000
50 IF ax< >29 THEN GOSUB 300
70 GOSUB 400
90 GOSUB 500
100 GOSUB 200: IF dead=0 THEN GOTO 50
```

```
10 CLS3:PRINT@266,"initializing";:SCREEN
   0,1
20 GOSUB 1000
25 GOSUB 1600
30 GOSUB 3000
50 IF AX< >29 THEN GOSUB 300
70 GOSUB 400
90 GOSUB 500
100 GOSUB 210:IF DD=0 THEN 50
```

The game is structured so that the graphics are defined and the high score is reset.

The main loop itself extends from Line 5Ø to Line 1ØØ, and continues all the time Freddy remains alive. The loop involves moving the arrow, if it has been fired, moving the spider, moving Freddy according to the key presses, and moving the balloons. Each of the relevant routines updates the variables.

FEEDING TIME

```
105 LET s(xinc)=1
110 FOR x=s(xpos) TO 29: GOSUB 500:
    NEXT x
120 LET s(yinc)=1: LET s(xinc)=0
125 FOR y=s(ypos) TO 19
130 IF y=my AND ax=29 THEN POKE
    23607,60: PRINT AT my+1,29;"□":
    POKE 23607,252
140 GOSUB 500
150 NEXT y
160 POKE 23607,60: PRINT AT 10,0; INK 2;
    PAPER 7; BRIGHT 1; FLASH 1;"You're
    dead! Another Game(Y/N)?";: POKE
    23607,252
165 LET a$=INKEY$: IF a$="" THEN GOTO
    160
170 IF a$="y" OR a$="Y" THEN GOTO 30
175 IF a$< >"n" AND a$< >"N" THEN
    GOTO 160
180 POKE 23607,60: CLS : STOP
```

```
105 S(XI)=1
110 FOR X=S(XP) TO 29:GOSUB 500:NEXT X
120 S(YI)=1:S(XI)=0
125 FOR Y=S(YP) TO 19
130 IF Y=MY AND AX=29 THEN PUT(232,
    (MY+1)*8)-(239,(MY+1)*8+7),S1,
    PSET
140 GOSUB 500
150 NEXT Y
160 FOR SL=180 TO 160 STEP -1:SOUND
    SL,1:NEXT:CLS:PRINT@256,"YOU'RE
    DEAD! ANOTHER GAME (Y/N)?"
165 A$=INKEY$:IF A$="" THEN 165
170 IF A$="Y" THEN 30
175 IF A$< >"N" THEN 160
180 CLS:END
```

After all the doors have been removed, dead (DD) is set to one, and Lines 1Ø5 to 18Ø are executed.

This routine moves the spider horizontally, until it is above the hapless Freddy, then vertically, chomping both him and the ladder. The player is then given the option of another go. In the Spectrum program, POKEing 6Ø into location 236Ø7 restores the character set pointer, so the full character set may be used.

INFLATION SOARS

```
210 LET b(count)=b(count)-1: IF b(count)
    < >0 THEN GOTO 280
220 LET b(count)=b(maxcount): PRINT AT
    b(ypos)+1,b(xpos);"□□";: LET
    b(ypos)=b(ypos)-1: IF b(ypos)=4
    THEN GOSUB 600: POKE
    23607,60:PRINT AT 1,10+(3-
    props)*9;"";AT 2,10+(3-
    props)*9;"□":POKE 23607,
    252: LET props=props-1
225 IF props=0 THEN LET dead=1
230 IF ((ay< >b(ypos) AND ay< >b
    (ypos)+1) OR (ax<b(xpos)-1 OR ax>b
    (xpos)+1)) THEN GOTO 250
240 LET score=score+b(points): GOSUB
    600: IF score>hiscore THEN LET hiscore=
    score: POKE 23607,60: PRINT AT 0,23;
    INK 0; PAPER 6;hiscore: POKE 23607,252
245 GOTO 380
250 GOSUB 4300
280 RETURN
```

```
210 B(CT)=B(CT)-1:IF B(CT)< >0 THEN
    280
220 B(CT)=B(MC)
221 B(YP)=B(YP)-1:IF B(YP)=4 THEN
    X2=B(XP)*8:Y2=B(YP)*8+8:PUT(X2,
    Y2)-(X2+15,Y2+15),SP,PSET:GOSUB
    600:X2=(10+(3-PP)*9)*8:PUT
    (X2,8)-(X2+7,15),S1,PSET:PUT(X2,
    16)-(X2+7,23),S1,PSET:PP=PP-1
225 IF PP=0 THEN DD=1
230 IF ((AY< >B(YP) AND
    AY< >B(YP)+1) OR (AX<B(XP)-1
    OR AX>B(XP)+1)) THEN 250
240 SC=SC+B(PO):X2=B(XP)*8:Y2=(B
    (YP)+1)*8:PUT(X2,Y2)-(X2+15,Y2+
    15),SP,PSET:GOSUB 600:IF SC>HS
    THEN HS=SC:GOSUB 1700
245 GOTO 400
250 GOSUB 4300
280 RETURN
```

b(count) (B(CT)) and b(maxcount) (B(MC)) are the most important elements of the balloon array. Each time the subroutine is called, Line 21Ø decrements b(count); when this reaches zero, the balloon will move. After the balloon has been moved, Line 22Ø copies the number in b(maxcount) into b(count). The balloon can be made to move at different speeds by simply varying the value of b(maxcount). Line 22Ø checks whether the balloon has been burst or if it has reached the top of the screen.

If the balloon has been burst, the score is

- COMPLETING THE GAME
- THE MAIN LOOP
- HAVING FREDDY FOR FOR BREAKFAST
- MOVING THE BALLOONS
- FIRING THE ARROWS
- UP AND DOWN THE LADDER
- ANIMATING THE MARTIAN SPIDER
- BURSTING BALLOONS

increased; if it has reached the top, one door is removed. If all the doors have been removed, dead is set to one.

TWANG!!

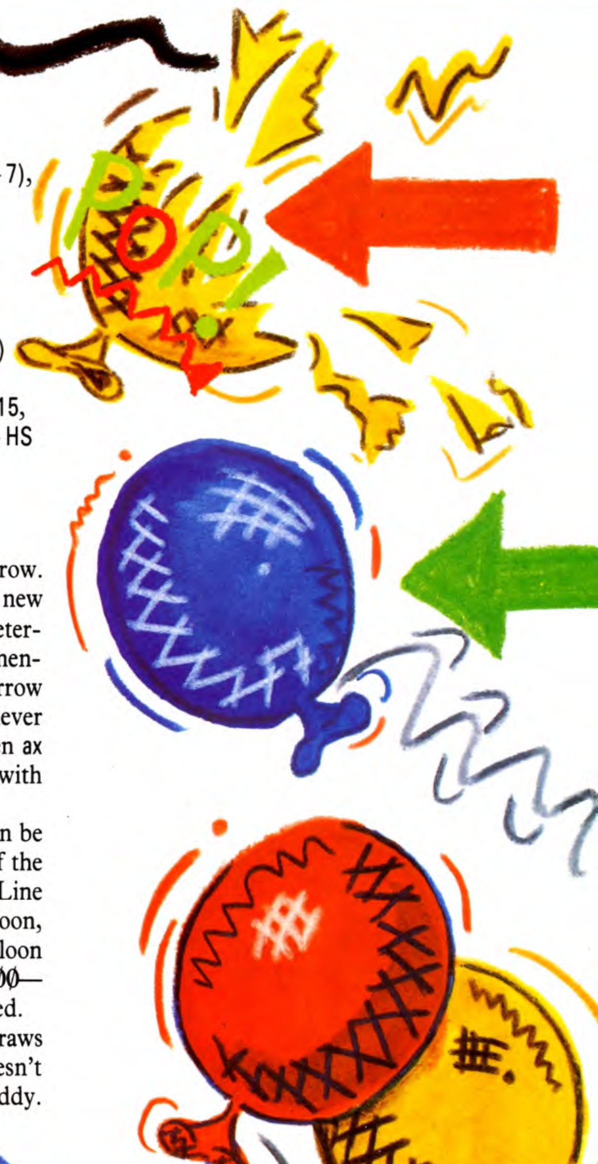
```
300 PRINT AT ay,ax;"□□": LET ax=ax-1:
    IF ax<Ø THEN LET ax=29: PRINT AT
    my+1,29; "e": LET ay=my+1: RETURN
310 IF ((ay=b(ypos) OR ay=b(ypos)+1)
    AND (ax=b(xpos) OR ax=b(xpos)+1))
    THEN LET score=score+b(points):
    GOSUB 6ØØ: IF score>hiscore THEN LET
    hiscore=score: POKE 236Ø7,6Ø: PRINT AT
    Ø,23; INK Ø; PAPER 6;hiscore: POKE
    236Ø7,252
330 IF ax<>29 THEN GOSUB 41ØØ
340 RETURN
```

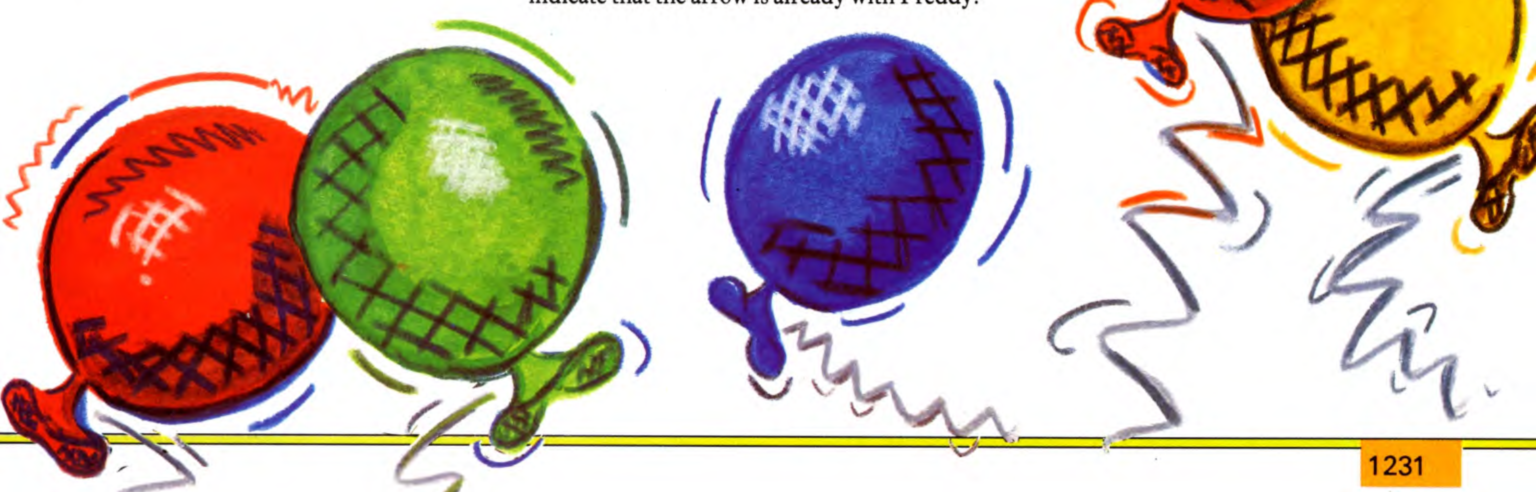
```
300 PUT(AX*8,AY*8)-(AX*8+15,AY*8+7),
    S2,PSET:AX=AX-1:IF AX<Ø THEN
    AX=29:PUT(232,(MY+1)*8)-(239,
    (MY+1)*8+7),E,PSET:AY=MY+1:
    RETURN
310 IF ((AY=B(YP) OR AY=B(YP)+1)
    AND (AX=B(XP) OR AX=B(XP)+1))
    THEN SC=SC+B(PO):X2=B(XP)*8:
    Y2=B(YP)*8+8:PUT(X2,Y2)-(X2+15,
    Y2+15),SP,PSET:GOSUB 6ØØ:IF SC>HS
    THENHS=SC:GOSUB 17ØØ
330 IF AX<>29 THEN GOSUB 411Ø
340 RETURN
```

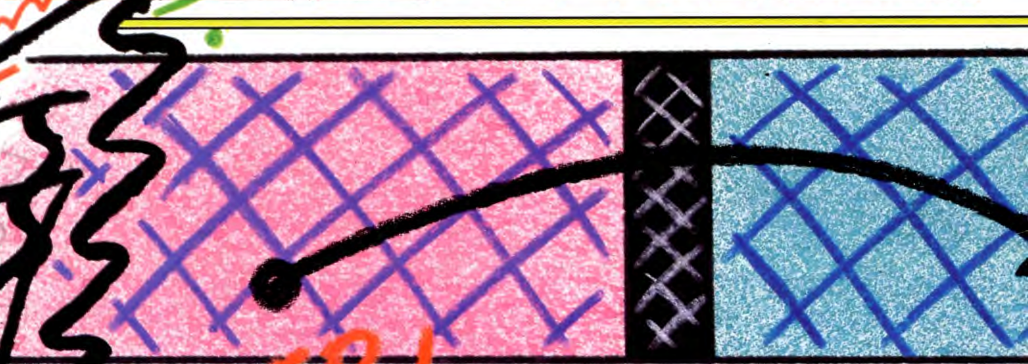
This is the routine which animates the arrow. The old image is blanked out, and the new one PRINTed at the next position—determined by variable ax(AX). ax is decremented in Line 3ØØ and, to prevent the arrow from being PRINTed off the screen, whenever ax falls below zero it is reset to 29. When ax has the value 29, the arrow is back with Freddy and can be fired using SPACE.

If the value of ax is 29, no balloons can be burst, so the subroutine is abandoned. If the arrow has been fired—ax<>29—then Line 31Ø checks if the arrow has hit the balloon, and increases the score if it has. If a balloon has burst, then the subroutine at Line 6ØØ—the balloon-bursting subroutine—is called.

Line 33Ø calls the subroutine which draws the arrow at Freddy's position, if ax doesn't indicate that the arrow is already with Freddy.







TAKING STEPS

```
400 LET a$=INKEY$: IF a$="" THEN
    RETURN
410 IF a$="Z" OR a$"z" THEN GOTO 45Ø
420 IF a$="c" OR a$="C" THEN GOTO
    44Ø
430 IF a$<>"□" THEN RETURN
432 IF ax<>29 THEN RETURN
434 LET ax=28: PRINT AT ay,29;"□":
    RETURN
440 IF my=19 THEN RETURN
445 PRINT AT my,3Ø; INK 6;"kl": LET
    my=my+1:: PRINT AT ay,29;"□": IF
    ax=29 THEN LET ay=ay+1
446 GOTO 47Ø
450 IF my=5 THEN RETURN
460 PRINT AT my+2,3Ø; INK 6;"kl": LET
    my=my-1: PRINT AT ay,29;"□": IF
    ax=29 THEN LET ay=ay-1
470 GOSUB 4ØØØ: RETURN
```

Tandy owners should change the 223 in Lines 41Ø, 42Ø and 43Ø to 247.

```
400 IF PEEK(337)=255 THEN RETURN
410 IF PEEK(341)=223 THEN 45Ø
420 IF PEEK(342)=223 THEN 44Ø
430 IF PEEK(345)<>223 THEN RETURN
432 IF AX<>29 THEN RETURN
434 AX=28:PUT(232,AY*8)-(239,AY*8+7),
    S1,PSET:RETURN
440 IF MY=19 THEN RETURN
445 PUT(24Ø,MY*8)-(255,MY*8+7),KL,
    PSET:MY=MY+1:PUT(232,AY*8)-(239,
    AY*8+7),S1,PSET:IF AX=29 THEN
    AY=AY+1
446 GOTO 47Ø
450 IF MY=5 THEN RETURN
460 PUT(24Ø,MY*8+16)-(255,MY*8+23),
    KL,PSET:MY=MY-1:PUT(232,AY*8)-
    (239,AY*8+7),S1,PSET:IF AX=29 THEN
    AY=AY-1
470 GOSUB 4ØØØ:RETURN
```

Lines 4ØØ to 43Ø read the keyboard. Lines 43Ø and 44Ø check if the space bar has been pressed, and then if Freddy has an arrow.

As Freddy moves up and down, ladder characters must be used to blank out either above or below him (or the ladder will disappear!), if ax equals 29, then the arrow must be moved, also.

SHORT, FAT, HAIRY LEGS

```
500 LET temp=s(xpos)+s(xinc)
510 IF temp<1 OR temp>8+(3-props)*9
    THEN LET s(xinc)=-s(xinc): GOTO 5ØØ
520 POKE 236Ø7,6Ø: PRINT AT s(ypos),
    s(xpos);"□□";AT s(ypos)+1,s(xpos);
    "□□": POKE 236Ø7,252
530 LET s(ypos)=s(ypos)+s(yinc): LET
    s(xpos)=temp: LET s(picture)=
    1-s(picture):GOSUB 42ØØ
540 RETURN
```

```
500 TE=S(XP)+S(XI)
510 IF TE<1 OR TE>8+(3-PP)*9 THEN
    S(XI)=-S(XI):GOTO 5ØØ
520 X2=S(XP)*8:Y2=S(YP)*8:PUT(X2,Y2)-
    (X2+15,Y2+15),SP,PSET
530 S(YP)=S(YP)+S(YI):S(XP)=TE:
    S(PI)=1-S(PI):GOSUB 42ØØ
540 RETURN
```

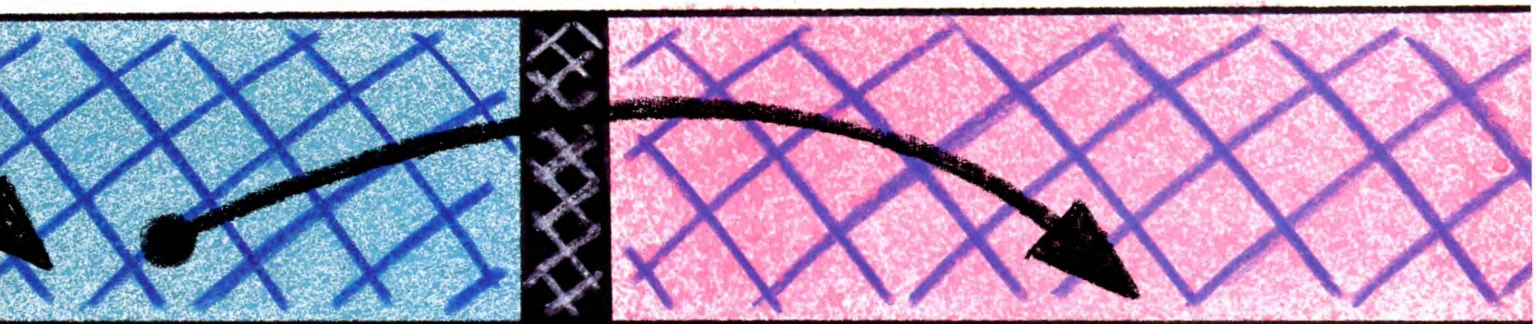
The last of the movement subroutines concerns the Martian spider. To make the game more interesting, it won't just sit there waiting for its nosh, but will pace up and down impatiently between the nearest door and the end wall. There are two spider pictures, the current picture number being stored in s(picture) (S(PI)) which is manipulated in Line 53Ø; either zero or one is produced.

Lines 5ØØ and 51Ø make sure that the spider doesn't escape from its cage before all the doors are removed.

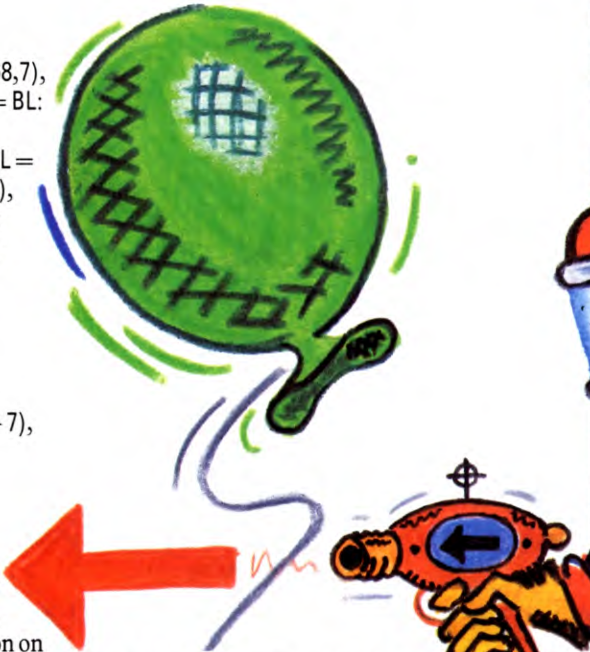
LIKE A LEAD BALLOON

```
600 PRINT AT b(ypos),b(xpos); BRIGHT 1;
    INK b(colour);"gh";AT b(ypos)+1,b(xpos);
    "ij"
610 POKE 236Ø7,6Ø
620 PRINT AT Ø,14; INK Ø; PAPER 6;score
630 BEEP .5,-2Ø
635 LET bl=bl-1: PRINT AT Ø,7; INK Ø;
    PAPER 6;bl;: IF bl=9 THEN PRINT INK Ø;
    PAPER 6;"□"
637 IF bl=Ø THEN LET bl=15+5*level: LET
    level=level+1: LET props=props-1:
    PRINT INK Ø; PAPER 6;AT Ø,7;bl;AT Ø,2;
    level: GOSUB 6ØØØ
640 PRINT AT b(ypos),b(xpos);"□□";AT
    b(ypos)+1,b(xpos);"□□"
650 PRINT AT ay,ax;"□□": LET ax=29:
    LET ay=my+1
660 POKE 236Ø7,252: GOSUB 4ØØØ: GOSUB
    5ØØØ: RETURN
```

```
600 X2=B(XP)*8:Y2=B(YP)*8:PUT
    (X2,Y2)-(X2+15,Y2+15),GJ,PSET
620 COLORØ:LINE(114,2)-(15Ø,7),PSET,BF:
    NU=SC:DRAW "C1;BM114,2":GOSUB
    165Ø
630 PLAY "V31;T2ØØ;O2;BAGFEDC;O1;
    BAGFEDC"
```



```
635 BL=BL-1:COLORØ:LINE(58,2)-(68,7),
    PSET,BF:DRAW "BM58,2;S2;C1":NU=BL:
    GOSUB 165Ø
637 IF BL=Ø THEN PP=3:LV=LV+1:BL=
    15+5*LV:COLORØ:LINE(14,2)-(24,7),
    PSET,BF:LINE(58,2)-(68,7),PSET,BF:
    NU=LV:DRAW "BM14,2;C1":GOSUB
    165Ø:GOSUB 6ØØØ:NU=BL:DRAW
    "BM58,2;C1":GOSUB 165Ø
64Ø X2=B(XP)*8:Y2=B(YP)*8:PUT
    (X2,Y2)-(X2+15,Y2+15),
    SP,PSET
65Ø PUT(AX*8,AY*8)-(AX*8+15,AY*8+7),
    S2,PSET:AX=29:AY=MY+1
66Ø GOSUB 4ØØØ:GOSUB 5ØØØ:
    RETURN
```



All that remains is to add a routine which will burst the balloon if an arrow is on target. The routine is very simple, just PRINTing the image of the burst balloon on screen, then erasing the image. The number of balloons remaining is adjusted, along with the level, if necessary. The arrow will be reset to Freddy's position, ready for him to fire at the next balloon.

Entering these routines will complete the game.

SETTING THE VARIABLES

```
3Ø B$="@A■■■■BC":AR$="DE":
    E$="FG■■■■HI":M$(1)=
    "LM■■■■NO":M$(2)=
    "PQ■■■■RS"
4Ø S$="■■TU■■■■VW■■■
    ■■■XY":L(1)=8:L(2)=2Ø:L(3)=32:
    L=Ø:M=Ø:D=1:BL=2Ø:BB=2Ø:
    SP=.3
5Ø Y$="■■■■■■■■■■■
    ■■■■■■■■■■■■■
    ■■":F$(1)=")■■)":F$(2)=
    "(■■(":LL=Ø
6Ø SY=7:SD$="■πJK■■■■JK
    ■■■■JK":KK=Ø:SC=Ø:F=Ø
7Ø GOTO 9Ø
```

These lines simply initialize the variables and strings needed.

THE BALLOON GOES UP

```
8Ø FOR Z=15 TO Ø STEP-1:POKE 646,Z
85 PRINTLEFT$(Y$,BY)SPC(BX)"■"E$:
    NEXT Z:IF KK=1 THEN KK=Ø:GOTO 98
9Ø L=L+1:PRINT"■■■■■";L
95 IF L=4 THEN L=3:LL=1:L(3)=L(3)+3
98 BX=INT(RND(1)*25)+1:BY=21
99 POKE 198,Ø:IF L=1 THEN 15Ø
125 PRINTLEFT$(Y$,BY)SPC(BX)"■■"B$
13Ø BY=BY-SP:IF BY<8 THEN PRINT
    "■■■■■"SPC(L(L)+3)F$(1):
    GOSUB 6ØØØ:GOTO 8Ø
135 PRINTLEFT$(Y$,BY)SPC(BX)"■■"B$
6ØØØ BL=BL-1:IF BL<Ø THEN BB=
    BB+5:BL=BB:SP=SP+.3
6Ø1Ø PRINT"■■"SPC(1Ø)BL"■■□":
    RETURN
```

There are four separate routines concerned with the balloons. Lines 8Ø and 85 are the balloon pop routine, which is used when an arrow is on target. Lines 9Ø to 99 set the level and the balloon location, Lines 125 to 135

animate the balloons as they float up the screen, and finally, Lines 6000 and 6010 start the next balloon (after popping or reaching the cage), and increments the speed of the balloon.

FREDDY

```
138 IF F=0 THEN PRINTLEFT$(Y$,SY+1)
    SPC(35)"□"
140 GET A$:IF A$="" THEN 180
150 IF A$="■" AND SY>8 THEN GOSUB
    4000:SY=SY-1
160 IF A$="■" AND SY<20 THEN
    GOSUB 4000:SY=SY+1
165 PRINTLEFT$(Y$,SY)SPC(36)S$
170 IF A$="□" AND F=0 THEN
    F=1:XX=34:YY=SY+1
180 IF F=0 THEN PRINTLEFT$
    (Y$,SY+1)SPC(35)"■ ■D"
900 GOTO 100
4000 PRINTLEFT$(Y$,SY)SPC(36)SD$:
    RETURN
```

Lines 138 to 180 move Freddy in response to key presses from the player. Lines 150 and 160 read the two cursor control keys, which are used to control upwards and downwards movement, and Line 170 checks if the space bar has been pressed. If it has been, an arrow is fired, and the fire flag F set.

If the player uses the cursor control keys to move Freddy, the man-blanking routine at Line 4000 is called so that he is animated.

STEP INTO MY PARLOUR

```
100 PRINT""SPC(M)M$
    (RND(1)*2+1)
103 IF M=36 THEN 3ØØØ
105 M=M+D:IF M<1 OR M>L(L)
    THEN D=-D:GOTO 1ØØ
11Ø PRINT""SPC(M)M$
    (RND(1)*2+1)
12Ø PRINT""SPC(L
    (L)+3)F$(RND(1)*2+1)
122 IF F=1 THEN 5ØØØ
123 IF LL=1 THEN 138
```

Lines 1ØØ to 12Ø animate the spider, making it wander along its cage. Line 122 calls the arrow animation routine if the fire flag has been set, and Line 123 makes the program jump over the balloon movement routine. By manipulating the value of LL, the speed of the balloon—or, rather, how often the balloon movement routine is called—can be regulated.

ALL OF A QUIVER

```
5ØØØ PRINTLEFT$(Y$,YY)SPC(XX)""
     AR$
5ØØ5 IF YY<2Ø THEN YY=YY+.1
5Ø1Ø XX=XX-1:IFXX<Ø THEN
     F=Ø:GOTO 123
5Ø2Ø PRINTLEFT$(Y$,YY)SPC(XX)""
     AR$
5Ø3Ø IF(XX=BXORXX+1=BX)AND(INT
     (BY)=INT(YY)ORINT(BY+1)=INT(YY))
     THEN 5Ø5Ø
5Ø4Ø GOTO 123
5Ø5Ø PRINTLEFT$(Y$,YY)SPC(XX)""
     AR$
5Ø55 F=Ø:KK=1:SC=SC+INT((26-YY)/
     2):PRINT""SPC(21)SC
5Ø6Ø GOSUB 6ØØØ:GOTO 8Ø
```

This routine animates the arrow by blanking out the last position, and reprinting it at the new position. The Commodore program also takes into account gravity.

FAST FOOD?

```
3ØØØ IF F=Ø THEN PRINTLEFT$(Y$,SY+1)
     SPC(35)"□"
3ØØ5 FOR Z=7 TO 21
3Ø1Ø PRINTLEFT$(Y$,Z-1)SPC(M)
     "JK"
3Ø2Ø PRINTLEFT$(Y$,Z)CHR$(153)""
     SPC(M)M$(RND(1)*2+1):FOR ZZ=1 TO
     2Ø:NEXT ZZ,Z
3Ø3Ø IF SC>HS THEN HS=SC
3Ø4Ø PRINT"YOU'RE DEAD!
     ANOTHER GAME(Y/N)?"
3Ø5Ø GET A$:IF A$="Y" THEN 2Ø
3Ø6Ø IF A$<>"N" THEN 3Ø5Ø
3Ø7Ø PRINT"":POKE53272,21:END
```

Lines 3ØØØ to 3Ø2Ø animate the spider, moving it down the ladder, eating Freddy. The remainder of the routine updates the high score if necessary, and offers another go.

Now complete the game by adding the following routines:

THE MAIN LOOP

```
1Ø MODE 1
2Ø PROCINITIALIZE
3Ø PROCGAMEINIT
4Ø REPEAT
5Ø IF AX%<>29 THEN PROCARROWMOVE
6Ø PROCMANMOVE
7Ø PROCSPIDERMOVE
8Ø PROCBALLOONMOVE
9Ø *FX21,Ø
1ØØ UNTIL DEAD%=1
```

The game is structured so that the graphics characters are defined and the high score is reset. These happen once only, before the next subroutine initializes a new game.

The main loop itself extends from Line 4Ø to Line 1ØØ, and REPEATs UNTIL Freddy dies (DEAD%=1). The loop involves calling PROCedures which move the arrow (if it has been fired), move the spider, move Freddy according to key presses, and move the balloons. Each of the relevant PROCedures updates the variables.

FEEDING TIME

```
11Ø S%(XINC%)=1
12Ø FOR X%=S%(XPOS%) TO 29:PROC
    SPIDERMOVE:NEXT
13Ø S%(YINC%)=1:S%(XINC%)=Ø
14Ø FOR Y%=S%(YPOS%) TO 19
15Ø IF Y%=MY% AND AX%=29 THEN
    PRINT TAB(29,MY%+1);"□"
16Ø PROCSPIDERMOVE
17Ø NEXT Y%
18Ø COLOUR1:PRINTTAB(Ø,1Ø); "YOU'RE
   DEAD! ANOTHER GAME (Y/N)?";
19Ø A$=GET$
2ØØ IF A$="Y" OR A$="y" THEN
    CLS:GOTO3Ø
21Ø IF A$<>"N" AND A<>"n" THEN
    19Ø
22Ø CLS:VDU23,1,1;Ø;Ø;Ø;:END
```

After all the bars have been removed, DEAD% is set to one, and Lines 11Ø to 22Ø are executed.

The routine moves the spider horizontally, until it is above the hapless Freddy, then vertically, chomping the ladder complete with Freddy. The player is then given an option on another go. The VDU 23 in Line 22Ø is used to put the cursor back on the screen.

INFLATION SOARS

```
23Ø DEFPROCBALLOONMOVE
24Ø B%(CNT%)=B%(CNT%)-1:IF B%
    (CNT%)<>Ø THEN ENDPROC
25Ø B%(CNT%)=B%(MAXCOUNT%)
26Ø PRINT TAB(B%(XPOS%),B%
    (YPOS%)+1);"□□";:B%(YPOS%)=B%
    (YPOS%)-1
27Ø IF B%(YPOS%)=4 THEN PROCBURST
    BALLOON:PRINT TAB(1Ø+(3-PROPS%)
    *9,1);"□"' TAB( 1Ø+(3-PROPS%)
    *9,2);"□":PROPS%=PROPS%-1:
    PROCMAKEBALLOON
28Ø IF PROPS%=Ø THEN DEAD%=1
29Ø IF((AY%<>B%(YPOS%) AND AY%<
    >B%(YPOS%)+1) OR (AX%<B%
    (XPOS%)-1 OR AX%>B%
    (XPOS%)+1)) THEN PROCDRAW
    BALLOON:ENDPROC
3ØØ SCORE%=SCORE%+B%(POINTS%):
    PROCBURSTBALLOON:PROCMAKE
    BALLOON:IF SCORE%>HISCORE% THEN
    HISCORE%=SCORE%:COLOUR13Ø:
    COLOURØ:PRINT TAB(23,Ø);HISCORE%:
    COLOUR 128:COLOUR2
31Ø ENDPROC
```

B%(CNT%) and B%(MAXCOUNT%) are the most important elements of the balloon array. Each time the PROCedure is called, Line 25Ø decrements B%(CNT%); when this reaches zero, the balloon will move. After the balloon has been moved, Line 26Ø copies the number in B%(MAXCOUNT%) into B%(CNT%). The balloon can be made to move at different speeds by simply varying the value of B%(MAXCOUNT%). Line 28Ø checks whether the balloon has reached the top of the TV screen, and it then calls PROCBURSTBALLOON if it has.

Line 3ØØ checks whether an arrow has connected with the balloon. If the balloon has been burst, the score is increased, and if the balloon reaches the top, one door is removed. If all the doors have been removed, DEAD% is set to one.

TWANG!!

```
32Ø DEFPROCARROWMOVE
33Ø PRINT TAB(AX%,AY%);"□□":AX%=
    AX%-1:IF AX%<Ø THEN AX%=29:
    AY%=MY%+1:COLOUR3:PRINT TAB
    (AX%,AY%);CHR$(132):ENDPROC
34Ø IF (AY%=B%(YPOS% OR AY%=B%
    (YPOS%)+1) AND (AX%=B%(XPOS%)
    OR AX%=B%(XPOS%)+1) THEN
    SCORE%=SCORE%+B%(POINTS%):
    PROCBURSTBALLOON:PROCMAKE
    BALLOON
35Ø IF SCORE%>HISCORE% THEN
```

```
    HISCORE%=SCORE%:COLOUR13Ø:
    COLOURØ:PRINT TAB(23,Ø);HISCORE%:
    COLOUR128:COLOUR2
36Ø IF AX%<>29 THEN PROCDRAWARROW
37Ø ENDPROC
```

This is the routine which animates the arrow. The old image is blanked out, and the new one PRINTed at the next position—determined by variable AX%. AX% is decremented in Line 33Ø and, to prevent the arrow from being PRINTed off the screen, whenever AX% falls below zero it is reset to 29. When AX% has the value 29, the arrow is back with Freddy and can be fired using SPACE.

If the value of AX% is 29, no balloons can be burst, so the PROCedure is abandoned. If the arrow has been fired—AX%<>29—then Line 36Ø checks if the arrow has hit the balloon, and increases the score if it has. If a balloon has burst, then PROCBURSTBALLOON is called.

Line 36Ø calls PROCDRAWARROW, which, in this case, draws the arrow at Freddy's position, if AX% doesn't indicate that the arrow is already with Freddy.

TAKING STEPS

```
38Ø DEFPROCMANMOVE
39Ø A$=INKEY$(Ø):IF A$="" THEN
    ENDPROC
4ØØ IF A$="Z" OR A$="z" THEN 44Ø
41Ø IF A$="A" OR A$="a" THEN 47Ø
42Ø IF A$<>"□" OR AX%<>29 THEN
    ENDPROC
43Ø AX%=28:PRINT TAB(29,AY%);"□":
    ENDPROC
44Ø IF MY%=19 THEN ENDPROC
45Ø COLOUR2:PRINT TAB(3Ø,MY%);CHR$
    (138);CHR$(139):MY%=MY%+1:PRINT
    TAB(29,AY%);"□":IF AX%=29 THEN
    AY%=AY%+1
46Ø PROCDRAWMAN:ENDPROC
47Ø IF MY%=5 THEN ENDPROC
48Ø COLOUR2:PRINT TAB(3Ø,MY%+2);
    CHR$(138);CHR$(139):MY%=MY%-1:
    PRINT TAB(29,AY%);"□":IF AX%=29
    THEN AY%=AY%-1
49Ø PROCDRAWMAN:ENDPROC
```

Lines 39Ø to 42Ø read the keyboard. Lines 42Ø and 43Ø check if the space bar has been pressed, and then if Freddy has an arrow.

As Freddy moves up and down (A moves up and Z moves down), ladder characters must be used to blank out either above or below him (or the ladder will disappear!), and if AX% equals 29, then the arrow must be moved, also.

SHORT, FAT, HAIRY LEGS

```
5ØØ DEFPROCSPIDERMOVE
51Ø TEMP%=S%(XPOS%)+S%(XINC%)
52Ø IF TEMP%<1 OR TEMP%>8+(3-
    PROPS%)*9 THEN S%(XINC%)=-S%
    (XINC%):GOTO51Ø
53Ø PRINT TAB(S%(XPOS%),S%(YPOS%));
    "□□":PRINT TAB(S%(XPOS%),S%
    (YPOS%)+1);"□□"
54Ø S%(YPOS%)=S%(YPOS%)+S%
    (YINC%):S%(XPOS%)=TEMP%:S%
    (PICTURE%)=1-S%(PICTURE%):PROC
    DRAWSPIDER
55Ø ENDPROC
```

The last of the movement PROCedures concerns the Martian spider. To make the game more interesting, it won't just sit there waiting for its nosh, but will pace up and down impatiently between the nearest door and the end wall. There are two spider pictures, the current picture number being stored in S%(PICTURE%) which is manipulated in Line 54Ø; either zero or one is produced.

Lines 53Ø and 54Ø make sure that the spider doesn't escape from its cage before all the doors are removed.

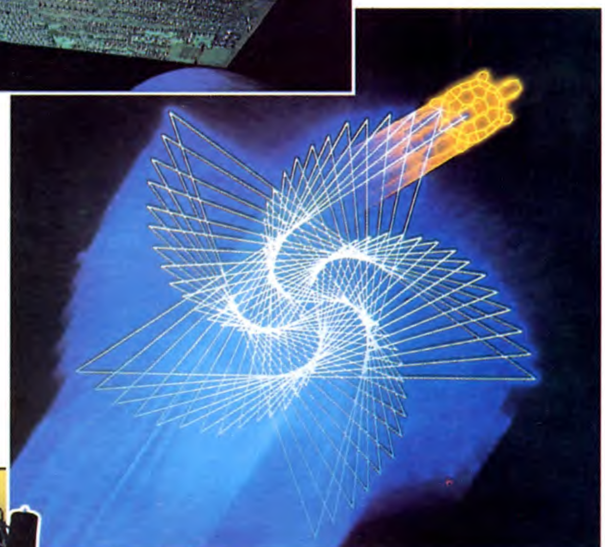
LIKE A LEAD BALLOON

```
56Ø DEFPROCBURSTBALLOON
57Ø COLOURB%(CLR%):PRINT TAB(B%
    (XPOS%),B%(YPOS%));CHR$(134);CHR$
    (135)' TAB(B%(XPOS%),B%(YPOS%)
    +1);CHR$(136);CHR$(137)
58Ø COLOUR13Ø:COLOURØ:PRINT TAB(14,Ø);
    SCORE%:COLOUR128:COLOUR2
59Ø SOUNDØ,-15,2Ø6,2:FORX=1TO7ØØ:
    NEXT
6ØØ BL%=BL%-1:COLOUR13Ø:COLOURØ:
    PRINT TAB(7,Ø);BL%;:IF BL%=9 THEN
    PRINT"□";
61Ø IF BL%=Ø THEN BL%=15+5*LEVEL%:
    LEVEL%=LEVEL%+1:PRINT TAB(7,Ø);
    BL%;TAB(2,Ø);LEVEL%:COLOUR128:
    COLOUR2:PROPS%=3:PROCDRAWDOORS
62Ø COLOUR128:COLOUR2
63Ø PRINT TAB(B%(XPOS%),B%(YPOS%));
    "□□";TAB(B%(XPOS%),B%
    (YPOS%)+1);"□□"
64Ø PRINT TAB(AX%,AY%);"□□":
    AX%=29:AY%=MY%+1
65Ø PROCDRAWARROW:PROCDRAWMAN
66Ø ENDPROC
```

All that remains now is to add a routine which will burst the balloon if an arrow is on target. The routine is very simple, just PRINTing the image of the burst balloon on screen, then erasing the image. The number of balloons remaining is adjusted, along with the level, if necessary. The arrow will be reset to Freddy's position, ready for the next balloon.

Coming soon in INPUT

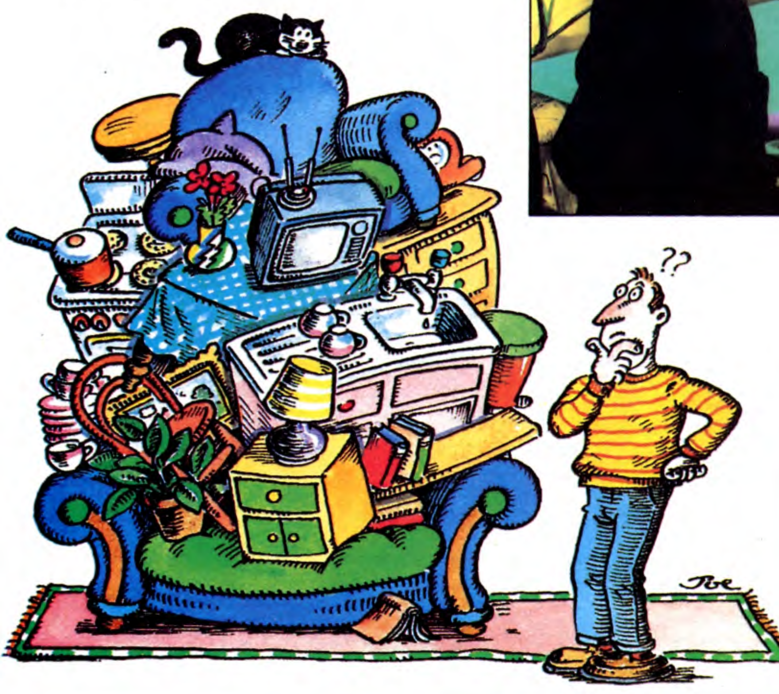
VOLUME 4



There's a bumper package of games, with the completion of CLIFFHANGER, a challenging WARGAME in which you use your strategy to outwit the computer— plus a new ADVENTURE that probes the limits of memory to create an exciting escapade set in a medieval castle.

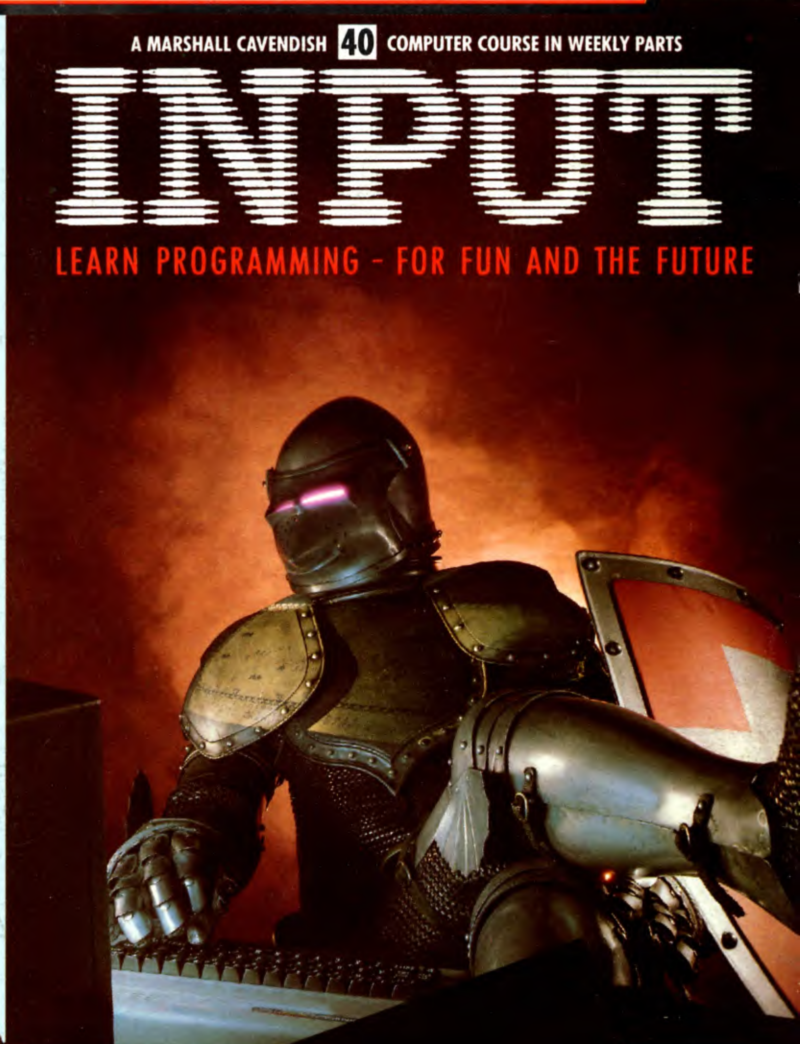
So far, most of the INPUT course has been about BASIC, because that is the language that is built into home micros. But now, there's a chance to look at some of the alternatives you can use on your computer, like PASCAL and FORTH— and why BASIC isn't always best.

And there's more on GRAPHICS, a new MUSIC program, a useful TOOLKIT, and details of COMPUTER LEARNING



ASK YOUR NEWSAGENT FOR INPUT

COMING IN ISSUE 40...



- *Starting a new series, INPUT looks at the alternatives to BASIC—programming in a NEW LANGUAGE. In the first part, discover what's involved in LEARNING LOGO*
- *Wargames aren't just for the belligerent—they are all about developing a strategy within the ground-rules of the game. The INPUT version takes you through a complete program*
- *Just for fun, enter the HOROSCOPE PROGRAM and predict the future*
- *There is cloudy weather in store for CLIFFHANGER as you add features to the sky above the playing area*
- *Explore the mysteries of growth with a MACHINE CODE LIFE GAME*
- *PLUS ... For COMMODORE users, how to combine COLOUR SPRITES*

ASK YOUR NEWSAGENT FOR INPUT