advanced sound & graphics for the Dragon computer

including machine code subroutines

keith & steven brain

advanced sound & graphics for the Dragon computer

including machine code subroutines

keith & steven brain

First published 1983 by:
Sunshine Books (an imprint of Scot Press Ltd.)
12–13 Little Newport Street,
London WC2R 3LD

Copyright © Keith and Steven Brain

ISBN 0 946408 06 8

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording and/or otherwise, without the prior written permission of the Publishers.

Cover design by Graphic Design Ltd.
Illustration by Stuart Hughes.
Typeset and printed in England by Commercial Colour Press, London E7.

Contents

		Page
Introduction		9
1	Sounding Off	11
2	Text and Low Resolution	27
3	High Resolution	37
4	Circles	51
5	DRAWing	67
6	Combining the Graphics Commands	77
7	On-Screen Movement	87
8	Copying the Screen	105
9	Graphic Presentation of Data	131
10	Three Dimensions	141
11	Rotation of Figures	147
12	Instant Keyboard Access to Hi-Res Commands	153
13	GETting and PUTting Hi-Res Characters	177
14	Working on a Grid	187
15	Animation	195
16	Sound Synthesis	213
17	Graphic MusicEditor	219
18	Beyond BASIC	231

Contents in detail

CHAPTER 1
Sounding Off
Play that tune: building up a note sequence, changing tempo, octave and note length, displaying the tune graphically and adding words to music. Improve your games with sound effects.

CHAPTER 2
Text and Low Resolution
Displaying characters, CLS, graphics characters, using PRINT TAB, PRINT USING, RESET, POINT and SCREEN commands.

CHAPTER 3
High Resolution
Setting up the hi-res screen, selecting the resolution and colours, setting the PMODE, choosing foreground and background colours, dealing with individual hi-resolution points.

CHAPTER 4
Circles
Circle is a versatile command which can produce many different types of curved shape including ellipses, arcs and spirals.

CHAPTER 5
Drawing
Experimenting with the 15 different DRAW commands to introduce features such as scale, colour, angles, move and blank move.

CHAPTER 6

Combining the Graphics Commands

The PIC-MAN demonstrates how to combine most of the hi-res drawing commands in a single program.

CHAPTER 7

On-Screen Movement

On-screen movement takes many forms. This chapter demonstrates a number of ways of moving low-res designs and combines them to move a more complicated picture of a starship around the screen.

CHAPTER 8

Copying the Screen

Copying whole graphics pages, storing and recreating displays, superimposing designs, selective erasing, storing screens and reproducing hard copies.

CHAPTER 9

Graphic Presentation of Data

Bar charts, line graphs, contour maps and pie charts.

CHAPTER 10

Three Dimensions

Presenting a three-dimensional view of an object is a very effective way of making it look more solid. How to plot in three dimensions. Experiment to form a 3-D box, tube and graph.

CHAPTER 11

Rotation of Figures

Angled draw commands: drawing at an angle, forming bisected triangles, rotating triangles and rectangles.

CHAPTER 12

Instant Keyboard Access to Hi-Res Commands

Drawing directly on the screen including single key routines, lines, rub-outs, circles, ellipses, GET and PUT, changing colours, painting, drawing with the joystick and labelling diagrams.

CHAPTER 13

GETting and PUTing Hi-Res Characters

Saving and using DRAW routines: transferring characters between programs, saving characters as machine code, dimensioning the arrays, GETting and PUTing characters.

CHAPTER 14

Working on a Grid

A grid system gives a very useful guide when you want to make sure your figure fits a particular format. Forming the grid, moving around, making a 'real' copy and saving it.

CHAPTER 15

Animation

Runner, Sprinter, Flying High, and Oasis

CHAPTER 16

Sound Synthesis

Some synthesiser features: repeating keyboard sound, changing the tempo, volume control, changing the octaves, and sound 'envelopes'.

CHAPTER 17

Graphic Music Editor

The editor allows you to enter a piece of music, display it in standard musical notation on the screen and then play it.

CHAPTER 18

Beyond BASIC

Exploring deeper inside your Dragon: POKEing into your program, changing the array names, hidden graphics modes, semigraphics, using machine code subroutines, partial PCLS and scrolling.

Introduction

The main aim of this book is to teach you how to make maximum use of sound and graphics in your own Dragon 32 programs. The Dragon has very useful sound and graphics capabilities and the Microsoft Color Basic employed contains some very powerful commands. However the very range and versatility of this implementation can be a barrier to the novice as it can all seem so complicated that he is not really sure where to start. The book therefore works from first principles, as to be able to develop impressive programs using these facilities you must understand very clearly both the basic manipulation of these commands and the best ways to use each of them. After the straightforward explanations of how to struggle with sound and grapple with graphics we get down to more detailed considerations of more complicated problems and the development of a series of useful tools and programs. Their value can perhaps be estimated by the realization that the figures and hi-resolution screen-copies for this book were created by the programs within it.

The basic format is that a command or an idea is taken and the routines built up step by step, exploring and comparing alternative possibilities wherever possible. Wherever relevant copies of the hi-resolution screen display are included, so that you can see what to expect. Rather than simply telling you what to do, and not to do, we encourage you to experiment with different approaches to let you see the results for yourself. As far as possible retyping of lines is strenuously avoided, but modification of lines is commonplace. All listings in this book are formatted to 32 columns so that they appear as you will see them on the screen, except that inverse characters appear as true lower case. In most cases spaces and brackets have been used liberally, to make the listings easier to read, but be warned that some spaces and brackets are essential so do not be tempted to remove them all.

All the routines have been rigorously tested and the listings have been checked very thoroughly so we hope that you will not find any bugs. It is a sad fact of life that most bugs arise as a result of 'tryping mitsakes' by the user. Semicolons and commas may look very insignificant but their absence can have very profound effects! If you do have problems with the listings in this book, or with your own programs, don't forget that the Dragon has a TRON function which allows you to follow execution of your programs, and that pressing SHIFT and @ will halt the execution of the program until you press another key.

If that doesn't solve the problem remember that variables are not reset until you RUN or use EDIT, so you can PRINT them out and see if they are legal values. Syntax errors (?SN ERROR) are usually the result of missing (or excessive) brackets, quotes or string signs ($), or the result of mis-spelt words, but they can also result if you remove essential spaces. Function call errors (?FC ERROR) usually arise because you are trying to use an illegal value in a function. Type mismatch (?TM ERROR) occurs if you mix up string and simple variables. The use of lots of strings can result in a out-of-string space error (?OS ERROR) which is easily cured by CLEARing extra string space. The maximum length of a string is 255 characters so if you try to add two long strings together you will get a ?LS ERROR. If you forget to DIMension an array, the array is too small, or you look for a negative value you will get a bad subscript error (?BS ERROR). If the program reaches a NEXT or RETURN without executing the relevant FOR or GOSUB you will get a ?NF ERROR or ?RG ERROR. Finally if you try to GOTO or GOSUB to a non-existent line you will get a ?UL error. This may be because you forgot to put a line in or you may have accidentally deleted a line.

The mythical dragons were colourful and noisy beasts and we hope that this book will help you to train your contemporary Welsh Dragon to display similar characteristics, without burning your fingers or too much midnight oil.

Keith and Steven Brain
Groeswen, July 1983

CHAPTER 1
Sounding Off

First Sounds

Everyone begins life with simple sounds but over the years their vocabulary builds up so why not learn about computer sound in the same way? We will start with the SOUND command which is the simplest way of creating sound on the Dragon. It needs two parameters, pitch and duration.

SOUND n,n

The first number is the pitch and can be any number between 1 and 255 inclusive. Pitch 89 is middle C on the piano.

The second number is the duration which can also be between 1 and 255 inclusive. A value of 16 for the duration is about one second.

For example:

```
SOUND 1,16
```

will produce a low note for one second and if we change the pitch to a higher value:

```
SOUND 255,16
```

will produce a high note for one second.

In the same way changing the second parameter:

```
SOUND 1,160
```

will produce a low note for ten seconds and

```
SOUND 255,160
```

will produce a high note for ten seconds.

Making such simple sounds is not that exciting so put the SOUND command inside a FOR . . . NEXT loop which will give every tone that can be accessed by Dragon sound commands in turn in ascending order.

```
10 FOR N=1 TO 255
20 SOUND N,1
30 NEXT N
```

If you reverse the loop a descending series is produced.

```
10 FOR N=255 TO 1 STEP -1
20 SOUND N,1
30 NEXT N
```

The duration can also be varied by the loop. If we now replace the duration of 1 by N the sounds will get longer and longer.

```
20 SOUND N,N
```

At this point you will probably try to stop the awful noise which is being produced as it seems as if it will go on for ever (255 gives a duration of about 16 seconds!). However you may be surprised to find that the BREAK key does not seem to work. In fact when any SOUND is being made the CPU is so busy that it cannot scan the keyboard and therefore the BREAK key will only work in the small gap between sounds.

If you want to relate duration to the N variable in the loop then you usually need to divide it down somewhat to get a sensible length. When you do this you must take care that you do not produce illegal values (less than 1). The simplest solution is to always add 1 to the calculated value.

```
20 SOUND N,N/20+1
```

Of course any STEP value can be used in the FOR-NEXT loop.

```
10 FOR N=255 TO 1 STEP -5
20 SOUND N,1
30 NEXT N
```

An uneven STEP will produce a more interesting SOUND.

```
10 FOR N=1 TO 255 STEP RND(5)
```

Even using FOR . . . NEXT loops you are restricted to a certain sequence but you could put the pitch and duration values in a DATA statement which could be READ back when required.

```
10 DATA 10,5,39,27,234,221,56,44,
37,11,33,24,75,64,31,53,20,24,48,
65
20 FOR N=1 TO 10
30 READ P,D
40 SOUND P,D
50 NEXT N
```

The above program will SOUND an odd assortment of notes and although you can use this method to play simple tunes a better way is to PLAY these. The SOUND command is therefore best left for making long notes or for playing set, related sequences.

Play that tune

The Dragon has a more powerful method of creating sound through the PLAY command, which has the syntax:

PLAY A$

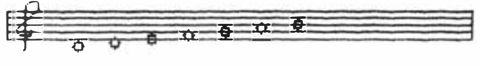
where A$ is a string up to 255 characters long. It allows you to define a whole string of notes and also gives you more control over the way these are played.

The easiest way to define which NOTEs to PLAY is to use the letters A-G to indicate the notes A-G. These are arranged in a set sequence (or scale for the musically minded). The scale of C is:

C D E F G A B

Compare this with the musical stave as shown in **Figure 1.1**.

Figure 1.1 Notes on the stave



To PLAY the scale of C you can type this as a direct command:

```
PLAY "C;D;E;F;G;A;B"
```

Note that the semicolons are optional and they are ignored when the string is PLAYed. If you leave them out you will hear no difference.

```
PLAY "CDEFGAB"
```

In general we tend to leave out the semicolons as they take up space, but it is sometimes useful to put some in as they can make sequences easier to read.

(Although it is also possible to define the notes by the numbers 1 to 12 this can get very confusing so forget that idea for the moment).

If you have the sheet music for a tune you can copy the notes into a string, but a number of other factors must also be taken into consideration. To illustrate the various parameters which must be taken into account let's look at entering the tune of a well known Christmas carol. The notes are as follows:

```
10 PLAY"FFCFGCAGABAGFFEDEFGAEDCC
CBABAGAFGEDCFFEFGFCAGABAGABAGFEF
BAGFF"
```

If you RUN and are very observant you might just recognise that the computer is trying valiantly to PLAY 'O come all ye faithful', but is somewhat flat! To correct this we need to indicate which notes must be flattened. Both flats and sharps are indicated by adding another character after the note. To FLATten notes you add a '−' after the letter for the note, and to SHARPen a note you add either a '+' or a '#' after the letter. The same tune with the appropriate notes flattened is now:

```
10 PLAY"FFCFGCAGAB-AGFFEDEFGAEDC
CCBABAGAFGEDCFFEFGFCAGAB-AGAB-AG
FEFB-AGFF"
```

The computer may catch you out if you add sharps and flats to notes at random, as it is cleverer than you and refuses to recognise B# or C− as they are not part of the 12 tone musical scale.

Even when you RUN this modified version you will notice that there is still something wrong. For a start the tune should change octave, at certain points, into a higher group of the notes A-G. ***Figure 1.2*** shows two octaves on a stave. To change octave you use:

On

where n is a number from 1–5. The default value is 2. For example:

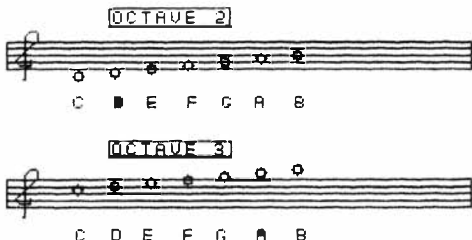
```
PLAY"CDEFGAB"
```

is different to

```
PLAY"O3CDEFGAB"
```

as the second scale is eight notes (an octave) higher.

Figure 1.2 Two Octaves



When you change the octave parameter the new octave will be used until you make another change, so you must remember to set it back where necessary. The amended tune is now:

```
10 PLAY"O2FFCFGCAGAB-AGFFEDEFGAE
DCCO3CO2BABAGAFGEDCFFEFGFCAGAB-A
GAB-AGFEFB-AGFF"
```

The O2 at the start is not strictly necessary in this case, as octave 2 is used by default. But it is a good idea to include it as otherwise complications can arise when you end the string in a different octave and then repeat the sequence. Enter these lines temporarily and notice the difference between the first time the string is played and the repeats.

```
100 PLAY"CDEFGABO3CDEFGAB"
110 GOTO 110
```

The first time through O2 is used for the first scale and O3 for the second, but O3 is then used for both scales on repeats. To repeat the first sequence every time add O2 to the front or back of the string.

```
100 PLAY"O2CDEFGABO3CDEFGAB"
```

or

```
100 PLAY"CDEFGABO3CDEFGABO2"
```

Meanwhile our carol is still not quite right as some of the notes are shorter than they should be, and some are too long. The Note Length Parameter (L) will solve this.

Ln

where n is any number from 1–255 and the default value is 4. As the number increases the length of the note decreases. L1 is a whole note, L2 a half note, L4 a quarter note etc. If you want to get intermediate values you can 'dot' the note. Adding a full stop after the number will make the value half as long again.

e.g. L2.

will be $\frac{1}{2}$ note + $\frac{1}{4}$ note = $\frac{3}{4}$ note.

The tune with the corrected note lengths is shown below.

```
10 PLAY"O2FL2FL4CFL2GCL4AGAB-L2A
L4GFL2FL4EDEFGAL2EL4DL8CL2CO3CO2
L4BAL2BAL4GAFGL4EL8DL4CFFEFGL2FL
4CAGAB-L2AL4GAB-AGFL2EL4FE-L2AL4
GL8FL2F"
```

Finally we can speed things up a bit by changing the Tempo parameter (T).

Tn

where n is a number from 1–255.

Note length sets the length of time each note is played for relative to the other notes, but Tempo alters the rate at which all the notes are played by the same proportion.

The default value for T is 2 so alter it to 5 and note that although the string is now played faster the note lengths are still in proportion.

```
10 PLAY"O2T5FL2FL4CFL2GCL4AGAB-L
2AL4GFL2FL4EDEFGAL2EL4DL8CL2CO3C
O2L4BAL2BAL4GAFGL4EL8DL4CFFEFGL2
FL4CAGAB-L2AL4GAB-AGFL2EL4FE-L2A
L4GL8FL2F"
```

We have already seen that we can loop around a string to repeat the tune, but what about a pause for breath at the end of each verse? The syntax of

Pause is like N and T but it just waits for the specified time (1 – 255) without making any sound. If we add PLAY a Pause before repeating we will create a gap between the verses.

```
20 PLAY"P50"
30 GOTO 10
```

If we have different groups of the congregation singing different verses perhaps we should alter the volume of each repeat. Of course you can change the volume by twiddling with your TV set but you can also let your Dragon control the volume, by using the Volume (V) parameter. The Volume command only differs from the other parameters in that it takes values from 1 to 31, instead of from 1 – 255, so it's no good trying to drown next door's stereo with a very high value. The default value is "V15" (half volume).

If we add

```
30 PLAY"V31"
40 GOTO 10
```

the first time the Volume will be half (V = 15) and on repeats the Volume will be full (V = 31).

If you want to repeat a particular musical sequence or set of commands it can be useful to define these as substrings which can then be eXecuted from the PLAY command with "X". The syntax is:

```
10 PLAY"XA$;"
```

where A$ is a valid musical string and the semi-colon is ESSENTIAL. You can execute and re-execute different phrases to make a whole tune. For example the tune for one verse of 'O come all ye faithful' could be defined as a string and then executed at three different Volumes and Tempos.

```
10 A$="O2FL2FL4CFL2GCL4AGAB-L2AL
4GFL2FL4EDEFGAL2EL4DL8CL2CO3CO2L
4BAL2BAL4GAFGL4EL8DL4CFFEFGL2FL4
CAGAB-L2AL4GAB-AGFL2EL4FB-L2AL4G
L8FL2F"
20 PLAY"T5V15XA$;T4V8XA$;T6V31XA
$"
```

Seeing what you are doing

It is possible to display the notes you are PLAYing if you define these as a string and not only PLAY this but also PRINT it. If you use only notes then life is very simple:

```
10 A$="CDEFGAB"
20 PLAY A$
30 PRINT A$
```

It would be nicer if you displayed each note as it was played, so let's slice up the string and PRINT each new note just before it is PLAYed.

```
20 FOR P=1 TO LEN(A$)
30 B$=MID$(A$,P,1)
40 PRINT B$;:PLAY B$
50 NEXT P
```

It is now easier to detect which, if any, of the notes is incorrect.

Of course 'real' strings to be PLAYed tend to be more complicated than this and, as not all the commands are of the same length, slicing the string can get more difficult. If we add sharps and flats and values for Volume, note Length, Tempo and Octave which are less than 10 we can still use only two positions in the string to define each command and then slice the string into two character sections. A little care must be taken in the way the characters are entered, as not all arrangements are acceptable. The rule on slicing is that the last character must NOT be a space, so that if a note is not sharp or flat it must be preceded by a space.

```
10 A$="T5O1F# G AB- E F C"
20 FOR P=1 TO LEN(A$) STEP 2
30 B$=MID$(A$,P,2)
```

Even longer commands can be accommodated (no spaces at the end!), although this tends to be very wasteful as lots of spaces must be inserted to pad each command out to the length of the longest command.

```
 10 A$="V31 T5 O1 F#  G  A B-  E
  F  C"
 20 FOR P=1 TO LEN(A$) STEP 3
 30 B$=MID$(A$,P,3)
```

and

```
 10 A$=" V31T100  O1   F#    G    A
  B-    E    F    C"
 20 FOR P=1 TO LEN(A$) STEP 4
 30 B$=MID$(A$,P,4)
```

To make it really easy to see what is going on when copying music or composing your own tunes we have designed a graphic music program which displays the music just as it appears on paper. This is described in detail later but don't be tempted to jump straight to it as you should study the graphics commands first so that you understand how it works.

Words and music

If you think for a moment about how we displayed the commands you will probably realise that instead of PRINTing the command which was being PLAYed we could PRINT something else instead, the obvious choice being the words to the tune. When you are adding the words you need to put the right syllable on the right note and also make sure that you do not put PRINT anything for commands which are not actual notes (e.g. changes in octave or note length). The syllables are put as DATA in lines 1 and 2, padding these out with spaces to give a neat display. We need to READ this DATA and print it only if the current command is a note and so the commands are sorted by INSTR which compares them with **X$**, a string containing all the notes used in the tune. When the next slice is not a note nothing is READ.

```
1 DATAO  ,COME ,ALL ,YE ,FAITH,FU
LL             ,.JOY,FULL ,AND ,TRI,U
M,PHANT                 ,O ,COME ,YE ,O
 ,CO,ME ,YE ,TO ,BE,TH,LE,HEM,CO
ME ,AND ,BE,HOLD ,HIM
   ,BORN ,THE ,KING ,OF ,A,N,GELS
         ,O ,COME ,LET ,US ,AD,O
RE ,HIM
2 DATACOME ,LET ,US ,AD,ORE ,HIM
            ,O ,COME ,LET ,US ,AD
,ORE ,H,IM              ,CHRI,ST ,THE
 ,LORD
10 X$="B- A B C D E F G"
20 CLS
30 A$="O2T5 FL2 FL4 C FL2 G CL4
A G AB-L2 AL4 G FL2 FL4 E D E F
G AL2 EL4 DL8 CL2 CP4L2O3 CO2L4
B AL2 B AL4 G A F GL4 EL8 DL4 C
F F E F GL2 FL4 C A G AB-L2 AL4
G AB- A G FL2 EL4 FB-L2 AL4 GL8
FL2 F"
40 FOR N=1TO LEN(A$) STEP 2
50 B$=MID$(A$,N,2)
60 X=INSTR(1,X$,B$)
```

```
70 IF X<>0 THEN READ C$:PRINT C$
80 PLAY B$
90 NEXT N
```

Sound effects

As well as being of great value for producing music SOUND and PLAY are also very useful for producing sound effects. Possible sound effects are many and varied, but very often use complex changes of parameters to achieve their effects.

These changes may be part of a preset sequence or may be linked to program variables. The commonest problems with sound effect generation are introducing program-linked changes and accidental generation of illegal parameter values.

SOUND is the easiest command to use as variables can be altered directly as described before, although the noise it makes is not very inspiring, unless it is repeated with changing parameters. A pitch of 0 is not allowed so take care that you cannot fall to that value. A simple way to prevent this is to always add 1 to the variable used in the SOUND command. This routine PRINTs and SOUNDs every pitch from 1 to 255 at random.

```
10 A=255
20 B=RND(255)
30 PRINT (A-B)
40 SOUND (A-B)+1,1
50 GOTO 20
```

You can link different SOUNDs to movement of each key if you scan INKEY$ and relate the pitch to the ASC11 value of the key. As SOUND needs simple variables you must first convert A$ to A.

```
10 A$=INKEY$:IF A$="" THEN 10
20 A=ASC(A$)
30 SOUND A,3
40 GOTO 10
```

The loop back if no key is pressed in line 10 is essential to prevent the crash which would occur if line 20 tried to take the ASC11 value of an empty string. Differences of 1 unit in pitch are not easy to detect, so why not multiply A by a factor to make differences between keys greater. The biggest factor which will not produce illegal values is 2 as 2*127=254.

PLAY is a more versatile command but introducing variables is a little more complicated because PLAY acts on a string. Any simple variable to be used must first be changed to string format with STR$ and then added to

the letter indicating the parameter to be varied. This will play a scale in a random octave.

```
10 A=RND(4)
20 PLAY"O"+STR$(A)
30 PLAY"CDEFGAB"
40 GOTO 10
```

As PLAY acts on a string you can make a simple tune directly on the keyboard byd PLAYing the contents of INKEY$.

```
20 A$=INKEY$
30 PLAY A$
40 GOTO 10
```

There is no need to check if A $ is empty as PLAYing an empty string is allowed. Of course you can only PLAY keys which correspond to notes without crashing. A simple way to ensure that only legal values are accepted is to use INSTR to compare the key pressed with another string (N$) containing a list of the valid keys.

```
10 N$="ABCDEFG"
30 IF INSTR(1,N$,A$)>0 THEN PLAY
 A$
```

Although Volume, Octave, Tempo and note Length can be altered by changing the actual values, as described above, it is sometimes convenient to use an alternative method which automatically steps the values up or down. To use this automatic method just add one of these SUFFIXES to the parameter:

+	adds one to the current value
−	subtracts one from the current value
>	multiplies the current value by two
<	divides the current value by two

Notice that the first two move slowly in steps of 1 unit but that the last two make more drastic changes as they double or halve the current value at each step. One point to watch with these commands is that it is very easy to reach illegal values and crash. For example:

```
100 PLAY"V>"
```

will double the default volume level (15) to almost full audio output (30), but:

```
100 PLAY"V>"
110 GOTO 100
```

will crash as a value of 60 has been calculated for the volume.

Increasing and decreasing volume is useful for indicating approach or retreat of something. For example this produces a police siren which gets louder as it approaches:

```
20 PLAY"L2T4FGV+"
30 GOTO 20
```

As it stands that starts from half volume and eventually crashes when V>31. To start from minimum volume define this separately outside the loop in line 10, and check in line 30 that not more than 30 loops have been made.

```
10 PLAY"V1"
30 L=L+1:IF L<31 THEN 20
```

If you want the approach to be more dramatic you can replace + with >, but if you start from V1 you must set V to the actual value of 31 for the fourth repeat as another V> will give 32.

```
10 PLAY"V1L2T4FGV>FGV>FGV>FGV>FG
V31FG"
```

Games programs usually call for lasers, phasers, zaps etc. and these usually use very short Tempo and/or Length and changes in Octave and Volume to produce the desired effect. Here are a few examples, but you can play for hours to produce even more impressive results!

```
10 PLAY"T255L255CDEFGAB":GOTO 10

(very fast ascending scale)
```

```
10 PLAY"T255L255CDEFGABAGFEDC":G
OTO 10

(ascending and descending scale)
```

```
10 PLAY"O1":FOR N=1 TO 4:PLAY"T2
55CDEFGABO+":NEXT N:GOTO 10

(increment octave)
```

```
10 PLAY"O1":FOR N=1 TO 4:PLAY"T2
55CDEFGABO-":NEXT N:GOTO 10

(decrement octave)
```

```
10 FOR N=1 TO 30:PLAY"V"+STR$(N)
+"O1T255DCDCDCDC":NEXT N

(increment volume)
```

```
20 FOR N=30 TO 1 STEP-1:PLAY"V"+
STR$(N)+"O1T255DCDCDCDC":NEXT N

(decrement volume)
```

Audio

The final sound function that we have not considered so far is AUDIO ON/OFF which allows you to connect and disconnect an external signal from the cassette input lead to the TV speaker. Using this you can play any type of sound track from the recorder. As the Dragon also allows you to turn the cassette MOTOR ON and OFF this means you can have total control of the playback. A simple application of this is in a totally unbiased computer controlled version of 'musical' chairs. If you put a music cassette in the recorder, press the play button, and RUN then the music will be turned on for a random length of time.

```
10 AUDIO ON:MOTOR ON
20 FOR N=1 TO RND(100000)+10000:
NEXT N
30 AUDIO OFF:MOTOR OFF
40 GOTO 10
```

Another use is to provide spoken instructions for a program whilst a demonstration is running. You need to record your program with CSAVE and then record your voice track behind it on the tape after a short gap. This sequence can be included as a program line with suitable prompts.

```
60000 CSAVE"name":MOTOR ON:FOR N
=1 TO 5000:NEXT N:MOTOR OFF
60010 CLS:PRINT"WHEN READY TO AD
D VOICE TRACK","PRESS A KEY"
60020 Q$=INKEY$:IF Q$="" THEN 60
020
60030 MOTOR ON:PRINT,,"RECORDING
",,"WHEN FINISHED PRESS A KEY"
60040 Q$=INKEY$:IF Q$="" THEN 60
040
60050 MOTOR OFF:PRINT,,"RECORDIN
G ENDED"
```

You now need to add MOTOR ON:AUDIO ON near the start of your program, and MOTOR OFF:AUDIO OFF when the track is ended.

```
10 CLS:PRINT"WELCOME",,"DO YOU W
ANT SPOKEN INSTRUCTIONS"
20 Q$=INKEY$:IF Q$="" THEN 20
30 IF Q$<>"Y" THEN 100
40 AUDIO ON:MOTOR ON
50 CLS:PRINT"TO STOP PLAYBACK PR
ESS A KEY"
60 Q$=INKEY$:IF Q$="" THEN 60
100 (rest of Program)
```

A more serious application is to link a spoken track to a learning program which tests spelling, vocabulary etc. You can then arrange for a word to be spoken whose correct spelling or translation must be entered. You could have a timing check (using the TIMER function) turning the MOTOR ON and OFF if all the spoken passages were more or less the same length. A difficulty with this is that the synchronisation of the program and voice track can go astray as the motor speed can be rather variable. The alternative is to make the user press a particular key to start and stop the tape.

```
100 AUDIO ON:MOTOR ON
110 PRINT "PRESS ENTER TO STOP T
APE",,,,
120 INPUT Q$:AUDIO OFF:MOTOR OFF
130 INPUT "ANSWER";A$
```

(checking routine)

```
200 GOTO 100
```

CHAPTER 2
Text and Low Resolution

The text screen of the Dragon contains 512 positions in a 32 by 16 matrix and the low-resolution graphics display gives you 2048 points which can be controlled individually on a 64 by 32 matrix. These use the same area of memory (addresses 1024 to 1535) so that text and low-res graphics can easily be mixed.

Characters

The alphanumeric and print control characters are defined by numbers from 0 to 127 and the numbers from 128 to 255 specify the graphics characters. Only some of these characters can be accessed directly from the keyboard. Some more characters can be selected using the CHR$ function and this program shows all the characters available using this method.

```
10 CLS
20 FOR N=0 TO 255
30 PRINT CHR$(N);
40 NEXT N
```

The most important extra characters available using the CHR$ function are the graphics characters (codes 128 to 255). These consist of eight sets of 16 blocks in which different segments are blacked out. The print control characters return blanks and they are not displayable.

There are still some more characters which can be displayed, but only if you directly change the contents of locations in the screen memory, as the BASIC interpreter rejects these. All the characters available on the Dragon can be displayed by adding the following lines which POKE the numbers 0 to 255 into screen memory.

```
50 FOR N=0 TO 255
60 POKE 1280+N,N
70 NEXT N
```

As location 1280 is equivalent to PRINT position 256 these characters lie below the first set. If you look closely at the two versions you will see that there are a number of differences. There is a whole extra line of inverse

symbols and numbers in the POKEd version and even when characters appear in both sections they are not always in the same place. You can use the extra inverse characters in your programs as long as you POKE them into place, and the differences in the order of the characters mean that you must be careful if you PEEK at the screen to detect what is at a particular point. **Table 2.1** compares the CHR$ and POKE values.

CLS

The text/low-res screen can be cleared to any of nine colours. The CLS command has the syntax:

CLS n

where n is a number from 0 to 8.

The codes for the nine colours are:

0— Black
1 — Green
2 — Yellow
3 — Blue
4 — Red
5 — Buff
6 — Cyan
7 — Magenta
8 — Orange

The default CLS, if no number is added, is green but you should realise that two green characters are available which look the same, but have different codes. CHR$(143) is a graphics character and CHR$(96) is a text space. Although CLS 1 to CLS 8 fill the screen memory with the appropriate coloured graphics character CLS on its own uses CHR$(96) rather than CHR$(143) which can sometimes cause confusion. To emphasise the point try entering CLS and CLS 1 as direct commands and then PRINT PEEK(1280) to see what the screen is filled with.

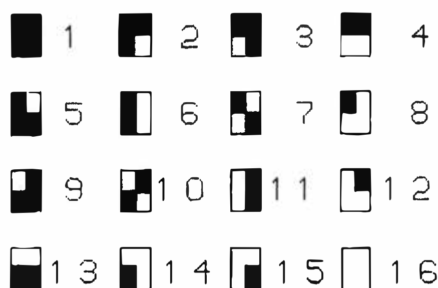
Graphics characters

As you have already seen codes 128 to 255 define the coloured graphics blocks, and there are 16 different characters **(Figure 2.1)** available in each colour. These blocks can easily be combined to build up a picture **(Figure 2.2)**.

```
10 CLS 0
30 PRINT CHR$(129);CHR$(135);C
HR$(131)
40 PRINT CHR$(128);CHR$(133);C
HR$(128)
50 PRINT CHR$(128);CHR$(142);C
HR$(138)
```

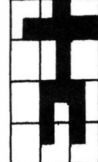
```
60 PRINT CHR$(128);CHR$(136);C
HR$(136)
```

Figure 2.1 Graphics characters



+127=GREEN
+143=YELLOW
+159=BLUE
+175=RED
+191=BUFF
+207=CYAN
+223=MAGENTA
+239=ORANGE

Figure 2.2 Forming a picture from graphics characters



As the sequence of blocks is the same for each colour the colour of a design can easily be changed by adding multiples of 16 to the codes for all

the characters included. The simplest way to do this is to add an increment variable (I) to each CHR$ number and then change this.

```
30 PRINT CHR$(129+I);CHR$(135+
I);CHR$(131+I)
40 PRINT CHR$(128+I);CHR$(133+
I);CHR$(128+I)
50 PRINT CHR$(128+I);CHR$(142+
I);CHR$(138+I)
60 PRINT CHR$(128+I);CHR$(136+
I);CHR$(136+I)
```

If you add this loop the program will show the same design in the next higher colour each time you press a key.

```
20 FOR N=0 TO 112 STEP 16
70 Q$=INKEY$:IF Q$="" THEN 70
80 NEXT N
```

PRINT @

The command PRINT @ allows you to define the PRINT position for the next character anywhere on the text screen. It has the syntax:

PRINT @ p,"message"

where p is a number between 0 and 512.

It can be used to position both text and graphics characters.

PRINT TAB

Although it is not mentioned in the manual a PRINT TAB command is available which allows you to displace the next PRINT position by a specified number from the present position. The format is:

PRINT TAB(d);"message"

where d is the displacement from the current PRINT position.

PRINT USING

This command allows you to control precisely the format of output to the screen, or a printer. It requires you to define the format you desire and then to give an 'output list' of items to be printed.

PRINT USING format;output list

There are many different ways of formatting output but in graphics programs we are only normally concerned with strings. The only relevant format command is therefore %. If you define the maximum length of a string to be printed as a number of spaces between two % signs then any string longer than this will have the end cut off to fit.

```
10 PRINT"WHAT IS YOUR NAME?"
20 INPUT A$
30 PRINT USING"%          %";A$
```

This can be useful where you want to prevent an unexpected value, or an INPUTed message, from upsetting a carefully planned screen display.

SET

The command SET turns the pixel defined by X and Y coordinates to a specified colour.

SET(X,Y,Colour)

The low-resolution screen is controlled in such a way that pixels of different colours cannot share the same character space, so that on any one character space there can only be one foreground colour and a black background, and in fact a pixel on the low-res screen is really only one quarter of a graphics character. As a demonstration try this:

```
10 CLS
20 SET(10,10,2)
```

The screen is first cleared to green and then you will see that one character space is now occupied by a black block with a yellow square in the top left hand corner. If you SET the point to the right, left, up or down to the same colour then the command works as expected, and the number of yellow blocks increases.

```
30 SET(9,10,2)
40 SET(11,10,2)
50 SET(10,9,2)
60 SET(10,11,2)
```

If you try to SET the point to the left to a different colour (3 = blue) this works fine:

```
40 SET(10,9,3)
```

but if you try to SET the point to the right of the first to a different colour then it doesn't.

```
50 SET(11,10,3)
```

Instead of producing alternate yellow and blue quarters this gives a completely blue top half of a character. The reasons for this are explained later, but for the moment just accept the fact that you cannot put a different colour on an adjacent pixel unless the boundary between them is a character boundary. In practice SET must always be used on a black screen, and you should try to avoid making lines adjacent wherever possible. These constraints mean that it is usually better to use hi-res for any detailed graphics work, although low-res still has the advantages of having nine colours available at once, and immediate access to text.

RESET

RESET is the converse of SET and turns a particular point off. It is used by specifying only X and Y coordinates, as the background colour in low-res is always black.

RESET(X,Y)

Notice that it is not possible to produce RESET by putting the colour 0 in SET, as the system rejects this.

POINT

POINT looks at a specified pixel and returns the colour code.

POINT(X,Y)

It is mainly used in comparisons in programs to check the state of a particular point before a decision is made:

```
100 IF POINT(X,Y)=6 THEN ..............
```

If the POINT tested is black then 0 is returned else if the POINT is coloured then one of the numbers 1 to 8 results. However if the POINT tests a position containing an alphanumeric character the answer is -1.

SCREEN Command

When the Dragon is turned on the text screen is automatically selected with black text on a green background, and when a program ends the system always falls back to this display. The actual definition of this state is SCREEN 0,0, where the first 0 indicates low-resolution and the second 0 indicates the first colour set. It is possible to change the colour set in low-res, so that text appears as red on orange by SCREEN 0,1, but this state only continues until the screen contents change or the program ends.

If you want to use this facility specify SCREEN 0,1 after PRINT and delay execution of the program.

```
10 PRINT"SCREEN 0,0"
20 Q$=INKEY$:IF Q$="" THEN 20
30 PRINT"SCREEN 0,1"
40 SCREEN 0,1
50 Q$=INKEY$:IF Q$="" THEN 50
60 GOTO 10
```

Now each time a key is pressed the colour set will change. Only text is changed and graphics characters appear as normal. The main use of SCREEN 0,1 is to highlight particularly important screen messages.

Table 2.1

TABLE SHOWING THE CHR$ AND POKE CODES FOR THE FIRST 128 CHARACTERS

CHARACTER	CHR$ CODE	POKE CODE
@ INV	-	0
a	97	1
b	98	2
c	99	3
d	100	4
e	101	5
f	102	6
g	103	7
h	104	8
i	105	9
j	106	10
k	107	11
l	108	12
m	109	13
n	110	14
o	111	15
p	112	16
q	113	17
r	114	18
s	115	19
t	116	20
u	117	21
v	118	22
w	119	23

x	120	24
y	121	25
z	122	26
[INV	123	27
INV	124	28
] INV	125	29
INV	126	30
INV	127	31
SPACE INV	–	32
! INV	–	33
" INV	–	34
# INV	–	35
$ INV	–	36
% INV	–	37
& INV	–	38
' INV	–	39
(INV	–	40
) INV	–	41
* INV	–	42
+ INV	–	43
, INV	–	44
- INV	–	45
. INV	–	46
/ INV	–	47
0 INV	–	48
1 INV	–	49
2 INV	–	50
3 INV	–	51
4 INV	–	52
5 INV	–	53
6 INV	–	54
7 INV	–	55
8 INV	–	56
9 INV	–	57
: INV	–	58
; INV	–	59
< INV	–	60
= INV	–	61
> INV	–	62
? INV	–	63
@	64	64
A	65	65
B	66	66

C	67	67
D	68	68
E	69	69
F	70	70
G	71	71
H	72	72
I	73	73
J	74	74
K	75	75
L	76	76
M	77	77
N	78	78
O	79	79
P	80	80
Q	81	81
R	82	82
S	83	83
T	84	84
U	85	85
V	86	86
W	87	87
X	88	88
Y	89	89
Z	90	90
[	91	91
	92	92
]	93	93
	94	94
	95	95
[SPACEBAR]	32	96
!	33	97
"	34	98
#	35	99
$	36	100
%	37	101
&	38	102
'	39	103
(	40	104
)	41	105
*	42	106
+	43	107
,	44	108
-	45	109

.	46	110
/	47	111
0	48	112
1	49	113
2	50	114
3	51	115
4	52	116
5	53	117
6	54	118
7	55	119
8	56	120
9	57	121
:	58	122
;	59	123
<	60	124
=	61	125
>	62	126
?	63	127

CHAPTER 3
High Resolution

Setting up the high-resolution screen

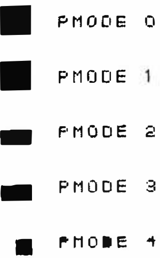
Before you can do any work in high-resolution you must set the system to a particular hi-res configuration and this means that you must make a number of decisions.

Resolution

First you must select the resolution (amount of fine detail) needed. Three different sizes of screen point are available (**Figure 3.1**) and these are in the followiig possible matrix arrangements.

128 horizontal by 96 vertical
128 horizontal by 192 vertical
256 horizontal by 192 vertical

Figure 3.1 Relative sizes of screen points



One screen point in the highest resolution is one quarter the size of one screen point in the lowest resolution, and half the size of one screen point in the middle resolution. Notice that in the middle resolution each point is a horizontally-elongated rectangle rather than a square.

Colours

The next consideration is the number of colours you want to use. Although a total of nine colours (well, eight colours plus black) are available on the Dragon you can only use these in hi-resolution in a restricted way. Only two colours or four colours can be used, and the colour mix is also fixed. The choice of number of colours is made by selecting one of the PMODE commands 0 to 4 (**Table 3.1**). Three two-colour and two four-colour possibilities are provided.

Table 3.1

RESOLUTION AND COLOURS

PMODE NUMBER	HORIZONTAL POINTS	VERTICAL POINTS	NUMBER OF COLOURS
0	128	96	TWO
1	128	96	FOUR
2	128	192	TWO
3	128	192	FOUR
4	256	192	TWO

Memory requirements and setting the PMODE

The amount of memory needed to support the hi-res display depends upon the PMODE selected, and ultimately on the amount of detail (resolution) and colour provided. When the Dragon is first switched on the area from

addresses 1536 to 6143 is automatically reserved for hi-res graphics, but your actual requirement may be less than this (**Table 3.2** and **Figure 3.2**). Memory is used in blocks of 1536 bytes, each of which is known as a 'graphics page'. At power-up four pages are reserved, but this can be increased by up to another four pages. The number of graphics pages reserved can be set by the user with the PCLEAR command. There is no point in reserving more memory than you need (as it is then unavailable for other purposes such as program or variable storage) so if you are using relatively low resolution use PCLEAR n to set-up only the required number of pages.

Table 3.2

MEMORY REQUIREMENTS IN DIFFERENT PMODES

PMODE	MEMORY (bytes)	MEMORY (pages)
0	1536	ONE
1	3072	TWO
2	3072	TWO
3	6143	FOUR
4	6143	FOUR

Figure 3.2 Arrangement of graphics pages

Address	Page
1536	1
3072	2
4608	3
6144	4
7680	5
9216	6
10752	7
12288	8

eg for PMODE 0 use PCLEAR 1

On the other hand more than four pages are needed for certain programming techniques (see later) so it may be necessary to PCLEAR a higher number of pages than four.

The PMODE command has two parameters. The first selects the PMODE and the second defines where in memory this PMODE is to be set up. The most usual position is page 1, but any reserved page can be specified. For example PMODE 0 takes up only one page of memory but this page can be any of the eight available pages. Of course you will get an error report if you try to use a page you have not reserved.

PMODE 0,1 = set up PMODE 0 on page 1
PMODE 0,8 = set up PMODE 0 on page 8

Higher PMODEs spread over more than one page but the second parameter still defines the start page.

PMODE 2,1 = set up PMODE 2 on pages 1 and 2
PMODE 2,2 = set up PMODE 2 on pages 2 and 3

It is possible to reserve different pages for different purposes at the same time, and these can also use different PMODEs. For example we could reserve five pages and set these aside as follows: a) one page for PMODE 0, two pages for PMODE 1, and two pages for PMODE 2 **(Figure 3.3)**.

```
10 PCLEAR 5:PMODE 0,1:PCLS:PMODE
 1,2:PCLS:PMODE 2,4:PCLS
```

Note that a PCLS command has been added after each PMODE command to ensure that each page is cleared. PCLS works like CLS on the text screen.

Colour sets and SCREEN

Once you have decided how many colours you need you must decide which particular colours to use. In each PMODE two alternative 'colour sets' are available and these are selected by the SCREEN command **(Table 3.3)**.

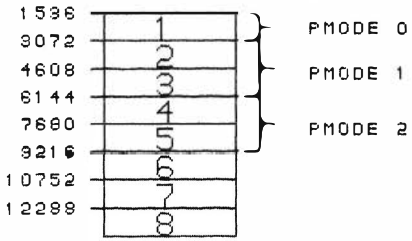
The SCREEN command needs two parameters, of which the colour set is the first. The second parameter tells the system whether to use the text or hi-res video memory. The number 1 selects hi-res. For example we will select colour set 0.

```
30 SCREEN 0,1
```

Table 3.3
COLOUR SETS

PMODE	COLOUR SET 0	COLOUR SET 1
0	black/green	black/buff
1	green/yellow/blue/red	buff/cyan/magenta/orange
2	black/green	black/buff
3	green/yellow/blue/red	buff/cyan/magenta/orange
4	black/green	black/buff

Figure 3.3 Setting different PMODEs on different pages



It is important to remember that it is the SCREEN command which actually switches the video display from one part of memory to another. If you insert a temporary line 15 GOTO 15 and RUN you will see that nothing seems to happen. Press BREAK, delete 15 and insert 100 GOTO 100 and RUN again. The normal text screen will now be replaced by the high-resolution screen. This hi-res screen will be displayed until all hi-res commands have been executed, when SCREEN automatically reverts to the text video memory (SCREEN 0,0). This happens automatically at the end of a program, hence the need for line 100 which forms an endless loop.

SCREEN always sets the video display according to the last PMODE specified. What we are actually seeing now is not page 1 but pages 4 and 5, as PMODE 2,4 was the last command. You can confirm that by inserting another PMODE command as line 20 to set the last page back to 2, when the screen will change to green.

```
20 PMODE 1,2
```

If you change the colour set in line 30 to 1 and RUN again the screen will appear as buff.

```
30 SCREEN 1,1
```

Foreground and background colours

The first named colour in **Table 3.3** is the background colour and the last named is the foreground colour. This means that the PCLS will clear the screen to the first colour, and any graphics command in which a colour is not specifically defined will automatically use the last colour. If you define a particular colour in a clear screen command by 'PCLS n' the screen will clear to that particular colour (n). But it is important to realise that this effect is only temporary and does not alter the actual foreground and background colours. If you want to alter foreground and background colours you must use the COLOR command which has the format:

COLOR(foreground,background)

Thus COLOR(1,2) will produce green on yellow and COLOR(4,3) red on blue. These settings will remain valid until they are deliberately altered.

Dealing with individual hi-resolution points

PSET

The simplest hi-resolution graphics command is PSET which turns on a single specified screen point. The format is:

PSET (X,Y,C)

and to use it you set the three parameters X, Y and C.

The first two are the X (horizontal) and Y (vertical) coordinates, which define the screen position which you want to alter. Although the number of individual screen points (resolution) varies with the PMODE that has been specified the coordinates for these commands always use a 256 by 192 matrix so that the centre of the screen is always at (128,96). The actual physical size of the point produced on the screen will of course depend upon the PMODE selected. As the same coordinates are used in all modes

you should not be surprised to find that using slightly different coordinates in the lower resolutions may produce the same result. For example in PMODE 0 there is no difference between these four commands, as they all turn on the top left hand screen point.

```
PSET(0,0)      PSET (0,1)
PSET(1,0)      PSET (1,1)
```

The last parameter is the number of the required colour (which must be a 'permitted' colour, that is a colour in the current set). If a colour is not specified then the current foreground colour is used. In practice if you want to use the current colour then the execution is actually quicker if you do not include the value for this in the PSET command.

You can see these effects quite clearly if you use these looping routines to PSET all the screen points in sequence under different conditions, and use the internal clock to check the time taken.

First in the highest-resolution (PMODE 4):

```
10 PMODE 4,1:SCREEN 1,0:PCLS
20 TIMER=0
30 FOR Y=0 TO 191
40 FOR X=0 TO 255
50 PSET (X,Y)
60 NEXT X
70 NEXT Y
80 PRINT "TIME TAKEN WAS";TIMER/
50;"SECONDS"
```

This will take about 237 seconds. If you now change the PMODE in line 10 to 0 and RUN the program again you will notice that the program seems to delay at the end of each line. In fact there is no delay and what you are seeing is the system PSETing the same points it has already PSET! As there is no value in PSETing the same point twice we might as well put STEP 2 on the end of lines 30 and 40, and see how much time that saves.

```
30 FOR Y=0 TO 191 STEP 2
40 FOR X=0 TO 255 STEP 2
```

Well, the result of around 60 seconds is much as you should expect when only one quarter of the work needs to be done.

The extra work involved when a colour parameter is included can be shown by changing line 50 so that colour 1 is specified.

```
50 PSET (X,Y,1)
```

The time needed rises steeply to 85 seconds, and the increase due to the change in defining the colour is actually even higher than it looks (50%), as 19 seconds are taken to complete this series of FOR-NEXT loops when they are completely empty. The moral should be clear — don't define parameters unless you have to!

Parameters can be derived with the command using the usual functions as shown in this demonstration which turns on points at random on the lowest hi-resolution screen:

```
10 PMODE 0,1:SCREEN 1,0:PCLS
20 PSET (RND(255),RND(191))
1000 GOTO 20
```

The routine is more interesting if a four-colour mode is selected and the colour also picked at random.

```
10 PMODE 1,1:SCREEN 1,0:PCLS
20 PSET (RND(255),RND(191),RND(4))
1000 GOTO 20
```

PRESET

The reverse of PSET is PRESET which sets the specified point to the background colour (that is turns it off).

If we PSET and PRESET rapidly we will produce a flashing point on the screen. Add the PRESETing line 60.

```
60 PRESET (X,Y)
```

Notice that there is no need to specify a colour with PRESET as the background colour is always used. In fact PSET can be made to do exactly the same job as PRESET if the background colour is used as the third parameter. This can sometimes be useful in programming, especially when the point colour to be used is calculated in the program. For example the routine below also produces a flashing point, but this time we will use the highest resolution PMODE 4.

```
10 PMODE 4,1:SCREEN 1,0:PCLS
50 FOR C=1 TO 0 STEP-1
60 PSET (X,Y,C)
70 NEXT C,X:GOTO 20
```

Notice that we must use a negative step so that colour 0 (background) is used last.

PPOINT

The final hi-res command which acts on a single screen point is PPOINT which finds the colour of the specified position. To show this working we will PSET a point at random and then check to see which point was PSET using PPOINT and two FOR NEXT loops to scan the screen.

```
10 PMODE 0,1:SCREEN 1,0:PCLS
20 X=RND(256)-1:Y=RND(192)-1
30 PSET (X,Y,1)
40 TIMER=0
50 FOR X=0 TO 255
60 FOR Y=0 TO 191
70 IF PPOINT (X,Y)<>1 THEN NEXT
Y,X
80 PRINT "POINT";X;Y;"SET"
90 PRINT TIMER/50;"SECONDS"
```

(notice that you need to use RND(256) – 1 to get numbers between 0 and 255 as RND(255) only gives 1 – 255).

You will observe that this routine is *very* slow! For example it takes 75 seconds to find a point PSET at 50,110 and 198 seconds to find a point PSET at 134,34. Of course we can speed things up rather by only checking 1 point in 4 by adding STEP 2 to the loops (as this is PMODE 0). That will now find a point at 68,156 in 28 seconds but it is still rather a painful process to examine the whole screen.

One of the main practical uses of PPOINT is collision detection in games programs. The reason that it is a realistic proposition in such cases is that here you only select a limited number of positions to check with PPOINT. For example if you have a yellow target and a red missile and you check for yellow at the missile coordinates you can detect contact. More experienced programmers amongst you may realise that you can actually avoid using PPOINT in this example if you keep a record of both target and missile coordinates and compare them directly. However this alternative is not possible where a record of the positions is not kept as variables.

PSET and PRESET are relatively slow and the more rapid and powerful LINE and CIRCLE commands can often be used instead. However PSET/PRESET are still important in certain applications, for example in plotting non-linear data and in producing a cursor to indicate screen position (see later).

Lines and boxes

PSET can be used to plot a series of adjacent points to form a horizontal line across the middle of the screen:

```
10 PMODE 1,1:SCREEN 1,0:PCLS
20 Y=96
30 C=2
40 FOR X=0 TO 255
50 PSET (X,Y,C)
70 NEXT X
1000 GOTO 20
```

If we add this PRESET routine then our line will next be 'undrawn' from the start.

```
80 FOR T=1 TO 1000:NEXT T
90 FOR X=0 TO 255
100 PRESET (X,Y)
110 NEXT X
```

On the other hand we could 'undraw' it from the other end with a decrementing FOR-NEXT loop.

```
90 FOR X=255 TO 0 STEP -1
```

Or make it dotted by only PRESETing certain points. We must remember, however, that in PMODE 1 points are set in pairs and we therefore need to use STEP 4.

```
90 FOR X=255 TO 0 STEP -4
```

If you want to use PSET to draw a line which is neither horizontal nor vertical then some calculations will be needed. To draw from 0,0 to 100,100 for example, is no problem as it is obvious that both X and Y must step by one for each point. However, even if you want to draw to a less regular point, it is still easy to calculate the appropriate step size by dividing the distance to be moved on the Y axis (YE-YS) by the distance to be moved on the X axis (XE-XS).

```
10 PMODE4,1:SCREEN1,0:PCLS
20 XS=0:XE=100
30 YS=0:YE=100
40 YI=(YE-YS)/(XE-XS)
30 FOR X=XS TO XE
40 PSET(X,Y)
50 Y=Y+YI
60 NEXT X
100 GOTO 100
```

Although you can use PSET in this way to draw straight lines it is actually much easier to use the LINE command. This is a very versatile command which only requires that you define the start and end points of the line in X,Y screen coordinates,and specify foreground or background colour with PSET or PRESET.

LINE(X1,Y1)-(X2,Y2),PSET

will draw a line from X1,Y1 to X2,Y2 in the foreground colour and

LINE(X1,Y1)-(X2,Y2),PRESET

will draw a line between the same coordinates in the background colour — that is it will actually erase the line.

Although the looping PSET routine described above works it is rather slow. A major advantage of using LINE is that the best fit between the path of the chosen line and the available screen points is automatically used without any user involvement, so that the routine is reduced to:

```
10 PMODE4,1:SCREEN1,0:PCLS
20 LINE(0,0)-(100,100),PSET
100 GOTO 100
```

This reduces the time required by a factor of about twenty from 0.86 secs to 0.04 secs. In practice this means that LINEs appear virtually instantaneously.

LINE is often used to connect a whole series of points to form graphs, or other complex figures. The points may be calculated in the program or stored in a DATA statement. In this example the DATA is READ into arrays which are then used in the LINE command. In general the LINE is drawn from the last point (X(N-1),Y(N-1)) to the next point (X(N),Y(N)) although a special arrangement must be made for the first line, where there is no last point. As Q is set to 0 in line 60 and reset to 1 in line 90 the first LINE will in fact be of zero length and drawn at X(N),Y(N).

```
10 PMODE4,1:SCREEN1,0:PCLS
20 DIM X(6),Y(6)
30 FOR N=1 TO 6
40 READ X(N),Y(N)
50 NEXT N
60 Q=0
70 FOR N=1 TO 6
80 LINE(X(N-Q),Y(N-Q))-(X(N),Y(N
),PSET
90 Q=1
```

```
100 NEXT N
110 DATA 1,1,10,10,255,14,190,6
0,12,80,45,180
```

If you want to draw a LINE in a colour other than the current background or foreground you must first redefine these parameters with the COLOR command. In this example start and end X and Y coordinates and the foreground colour (C) are all chosen at random, and then COLOR is used to ensure that the line is drawn in the chosen colour.

```
10 PMODE3,1:SCREEN1,0:PCLS
20 FOR N=1 TO 20
30 X1=RND(256)-1:Y1=RND(192)-1
40 X2=RND(256)-1:Y2=RND(192)-1
50 C=RND(3)+1
60 COLOR C,1
70 LINE(X1,Y1)-(X2,Y2),PSET
100 NEXT N
110 GOTO 110
```

The LINE command can also easily be used to produce rectangular boxes by adding the suffix B and specifying the coordinates of the top left hand corner and the bottom right-hand corner.

LINE(X1,Y 1)-(X2,Y2) ,PSET,B

If we merely add this suffix to line 70 of the last program we will generate different coloured boxes instead of simply sloping lines.

```
70 LINE(X1,Y1)-(X2,Y2),PSET,B
```

As usual PRESET will erase the box.

There are two ways of filling in the area contained within the box. The simplest method is to use the suffix BF (for box filled) which automatically fills the box with the foreground colour.

```
70 LINE(X1,Y1)-(X2,Y2),PSET,BF
```

This will always fill the box with the same colour as that used for the outline. If you want the outline and contents to differ you can draw a filled box slightly smaller than required and then draw an empty box around this in a different colour. Notice that 2 must be added to or subtracted from the X coordinates as this is PMODE 3 and points are set in pairs.

```
50 C=RND(2)+1
70 LINE(X1+2,Y1+1)-(X2-2,Y2-1),P
SET,BF
80 COLOR 4,1
90 LINE(X1,Y1)-(X2,Y2),PSET,B
```

PAINTing

An alternative method of filling the box is to use the PAINT command which will fill any specified area with any permitted colour. The start coordinates must be specified, followed by the colour to be used for PAINTing. and the final parameter is the 'border' colour which tells the system when to stop PAINTing.

For example:

PAINT(128,96),4,2

means start at, the screen centre (128,96) and turn all points to colour 4 (red) until you reach points which are colour 2 (yellow), or if there is nowhere left to move then stop.

```
70 LINE(X1,Y1)-(X2,Y2),PSET,BF
80 PAINT(X1+2,Y1+1),4,2
```

A comparison of the speed of producing a filled box at (50,50)-(150-150) by each method shows that LINE ... BF is more than twice as fast as PAINT (0.8 secs as against 1.7), but against this must be set the greater ease of specifying the PAINT colour,and the fact that it will fill irregular areas as easily as boxes. For example try:

```
10 PMODE3,1:SCREEN1,0:PCLS
20 LINE(10,10)-(80,20),PSET
30 LINE(80,20)-(95,60),PSET
40 LINE(95,60)-(25,90),PSET
50 LINE(25,90)-(10,10),PSET
60 PAINT(12,12),2,4
```

Any point within the chosen area can be used as the start position, although a start point near a corner gives a smoother effect as otherwise filling does not always seem to proceed in a logical fashion.

A couple of problems often crop up when using PAINT:

1) Nothing happens

Remember that if the start point is already set to the border colour then the command will end as soon as it starts. Thus this command will have no effect:

```
60 PAINT(12,12),2,1
```

2) Paint leaks into unwanted places

a) Remember that this paint is very corrosive and will easily leak through any minute 'pinhole' in the border. Make this very small modification to line 50 and watch the disastrous consequences. Once started PAINTing cannot be stopped, even the BREAK key having no effect.

```
50 LINE(25,90)-(10,9),PSET
```

This potential problem can sometimes be turned to advantage if you want to PAINT a series of adjacent independent areas the same colour, if you can deliberately form some leakage points in strategic places.

```
10 PMODE3,1:SCREEN1,0:PCLS
20 LINE(10,10)-(100,40),PSET,B
40 LINE(30,40)-(80,80),PSET,B
60 PAINT(12,12),2,4
```

This program will produce two adjacent boxes but only the top one will be PAINTed. To PAINT both at once we must provide a leakage point.

```
50 PSET(40,40,3)
```

b) Remember that you can only specify one border colour so that PAINT cannot be used to fill an area which is bordered by different colours. If the COLOR is changed between forming the two boxes the outlines will differ in colour (first box red and second blue).

```
30 COLOR 3,1
```

If you now try to PAINT these everything except three walls of the first box will be turned yellow.

CHAPTER 4
Circles

CIRCLE is a very versatile command which can be used to produce many different types of curved shape. In its simplest form it needs only three parameters X, Y and R:

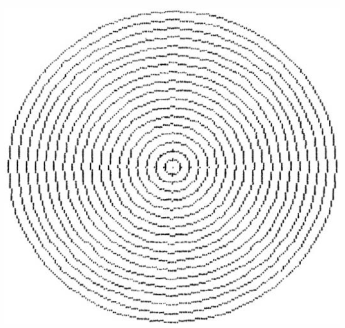
CIRCLE (X,Y),R

X and Y are the screen coordinates of the centre of the circle, and R is the radius in screen points (on a 256 by 192 matrix).

A stepped FOR-NEXT loop can be used to produce a series of concentric circles (**Figure 4.1**) of increasing radius.

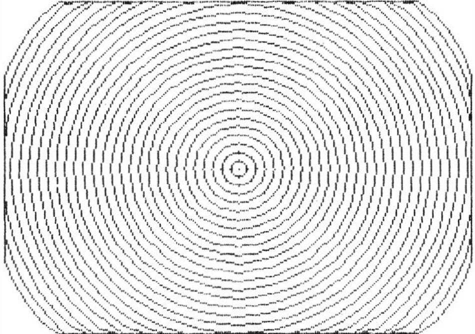
```
20 PMODE 4,1:SCREEN 1,0:PCLS
40 FOR R=0 TO 90 STEP 5
50 CIRCLE (128,96),R
60 NEXT R
200 GOTO 200
```

Figure 4.1 Concentric circles



If very large CIRCLES are drawn these will fall outside the screen area and will appear flattened (**Figure 4.2**). As the number of points on the Y axis is 192 the largest undistorted circle which can be drawn has a radius of 96 screen points.

Figure 4.2 Circles falling outside screen area



If the STEP in the FOR-NEXT loop is negative the circles will diminish in size instead.

```
40 FOR R=90 TO 0 STEP -5
```

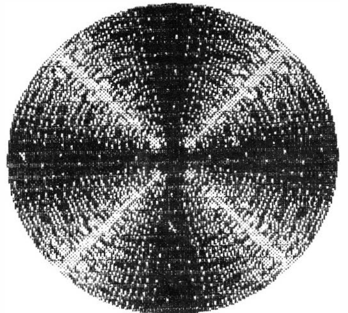
You might expect that a STEP of 1 (default) would give you a completely filled circle, but in practice that does not happen (**Figure 4.3**) and some areas are left blank.

```
40 FOR R=0 TO 90
```

The reasons for these gaps are the approximations which must be made in fitting the mathematical calculations of the circle circumference to available screen points. Close examination reveals that no screen circle is ever completely round but is made up of a combination of short straight lines. How perfect the circle looks depends on the resolution used. Most of our examples use PMODEs 3 and 4, but it is worth comparing the display in the various PMODEs (**Figure 4.4**).

```
10 CLS:INPUT"PMODE";P
20 PMODE P,1:SCREEN 1,0:PCLS
40 FOR R=0 TO 90 STEP 10
50 CIRCLE (128,96),R
60 NEXT R
70 I$=INKEY$:IF I$="" THEN 70 EL
SE 10
```

Figure 4.3 'Filled' circles



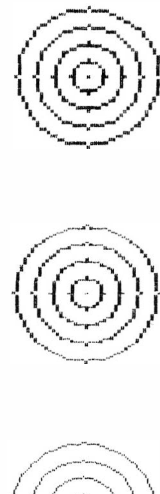
If nothing else is specified then the current foreground colour will be used to draw the circle, but particular colours can also be defined by the fourth parameter C.

CIRCLE (X,Y),R,C

This is often used to selectively erase CIRCLES by drawing them in the background colour.

```
20 PMODE 4,1:SCREEN 1,0:PCLS
30 FOR C=1 TO 0 STEP-1
40 FOR R=0 TO 90 STEP 10
50 CIRCLE (128,96),R,C
60 NEXT R
70 NEXT C
200 GOTO 200
```

Figure 4.4 Display in different PMODEs (from top 0,2,4)

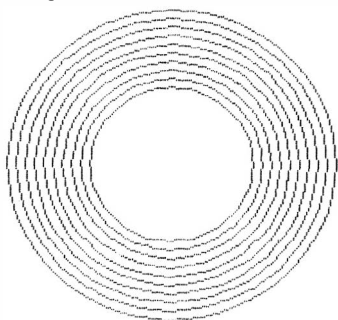


As it stands this will draw green circles of increasing radius and then black circles of increasing radius (**Figure 4.5**) finally leaving a blank screen.

If you move the position of the FOR C. . . .NEXT C loop inside the radius loop you will produce flashing circles. Delete lines 30 and 70 and rewrite them as 45 and 55.

```
40 FOR R=0 TO 90 STEP 10
45 FOR C=1 TO 0 STEP-1
50 CIRCLE (128,96),R,C
55 NEXT C
60 NEXT R
```

Figure 4.5 Erasing circles



Or you could reverse the STEP and erase the circles from the outside in.

```
30 C=1
40 FOR R=0 TO 90 STEP 10
50 CIRCLE (128,96),R,C
60 NEXT R
70 C=0
80 FOR R=90 TO 0 STEP -10
90 CIRCLE (128,96),R,C
100 NEXT R
```

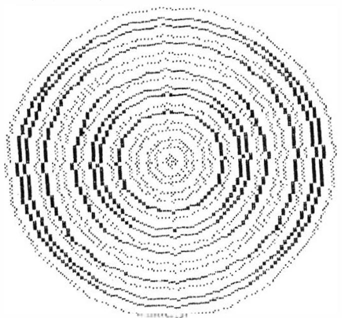
Colour can be chosen at random (**Figure 4.6**):

```
20 PMODE 3,1:SCREEN 1,0:PCLS
40 FOR R=0 TO 90 STEP 5
50 CIRCLE (128,96),R,RND(4)
60 NEXT R
200 GOTO 200
```

Or it may be calculated in some way. For example we could relate the colour to the circle radius (always remembering that the result must be a valid colour).

```
40 FOR R=20 TO 80 STEP 5
50 CIRCLE (128,96),R,R/20
```

Figure 4.6 Coloured circles



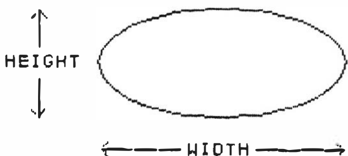
Ellipses

Although the command is called CIRCLE it is just as easy to use it to draw ellipses, by changing the next parameter, HW, the height/width ratio.

CIRCLE (X,Y),R,C,HW

The height/width ratio is simply the height of the 'circle' divided by its width (**Figure 4.7**). For a real circle the value is I. If the design is short in relation to its width HW will be less than I, and if it is tall HW will be greater than 1.

Figure 4.7 H/W ratio of ellipse



When changing any of these later parameters you must always take care to include also all the earlier parameters, or chaos will reign. You do not have to put actual numbers but you must at least include the commas which indicate where parameters start and end.

CIRCLE (128,96)90,1,0.5

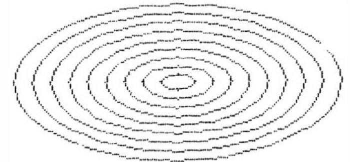
or CIRCLE (128,96),90,,0.5

In all these examples we have written non-integer numbers in full for clarity, but in practice they can be abbreviated as the leading zero is not essential.

A series of concentric ellipses with the same HW can be produced as easily as circles **(Figure 4.8).**

```
20 PMODE 4,1:SCREEN 1,0:PCLS
40 FOR R=10 TO 90 STEP 10
50 CIRCLE (128,96),R,1,0.5
60 NEXT R
200 GOTO 200
```

Figure 4.8 Concentric ellipses



If the radius is kept constant but HW varied from 0 to 1 a series of flattened ellipses of equal diameter are formed **(Figure 4.9).**

```
40 FOR HW=0 TO 1 STEP 0.1
50 CIRCLE (128,96),90,1,HW
60 NEXT HW
```

If HW is increased further vertically distortion occurs. **Figure 4.10** shows the next ten ellipses from HW 1 to 2 in 0.1 steps. Although HW can be any value up to 255 large values are not used as they simply give vertical lines.

The HW ratio of a normal television set is 4/3 and the screen display is 256 by 192 so that the largest ellipse which can be accommodated has a radius of 128 and a HW ratio of 0.75 **(Figure 4.11).** This type of design in the screen corners could form a nice 'vignette' setting for your 'golden oldies'.

```
40 FOR R=125 TO 180 STEP 3
50 CIRCLE (128,96),R,1,0.75
60 NEXT R
```

Figure 4.9 Changing H/W ratio from 0 to 1

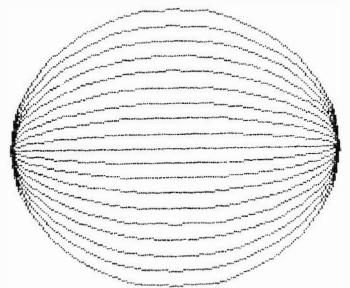


Figure 4.10 Changing H/W ratio from 1 to 2

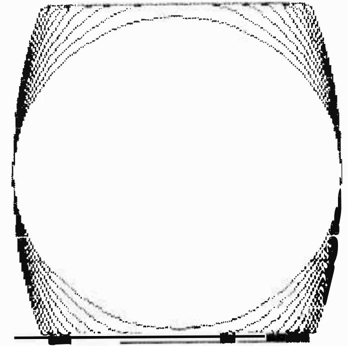
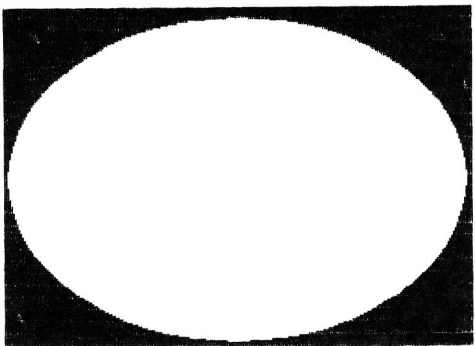


Figure 4.11 Vignette effect



Arcs

So far we have always drawn complete circles, but we can also draw limited parts (ARCs) of circles (and ellipses). The last two possible parameters define the start (S) and end (E) of the circle.

CIRCLE (X,Y),R,C,HW,S,E

CIRCLE always draws in a clockwise direction from the three o'clock position. The 3 o'clock position is defined as 0 and points on the circumference are increasing values between 0 and 1.

If we start at 0.5 and end at 1 the top half of a circle is formed **(Figure 4.12)**.

```
40 FOR R=10 TO 90 STEP 10
50 CIRCLE (128,96),R,1,1,0.5,1
60 NEXT R
```

A start of 0.5 and end of 0.75 gives the top left quadrant **(Figure 4.13)** and similarly S = 0.75 and E = 1 gives the top right quadrant **(Figure 4.14)**.

```
50 CIRCLE (128,96),R,1,1,0.5,0.7
5
```

Figure 4.12 Semi-circle

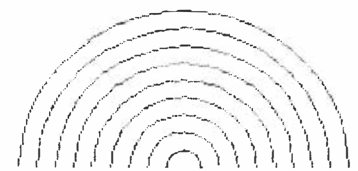
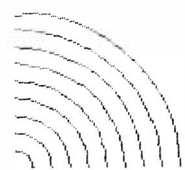


Figure 4.13 Top left quadrant

Figure 4.14 Top right quadrant



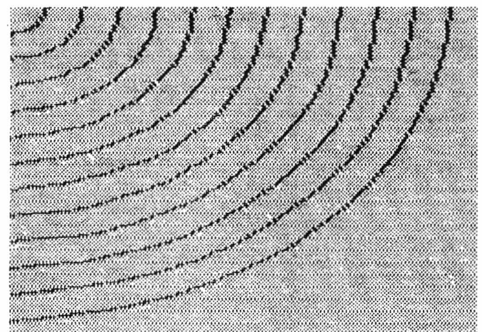
Smaller differences between S and E will draw smaller segments and these can start and end at any point (**Figure 4.15**). Partial ellipses can also be drawn in the same way.

You can draw an undistorted arc on the screen even if the full circle from which it comes would be so large that it would overlap the screen, provided that the centre of the circle is a valid screen point (**Figure 4.16**).

```
40 FOR R=20 TO 240 STEP 20
50 CIRCLE (0,0),R,1,0.75,0,0.25
60 NEXT R
```

Figure 4.15 Segment

Figure 4.16 Arcs of circle with centre just on screen

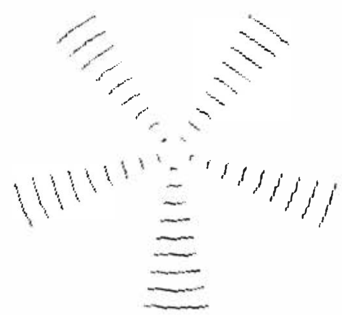


If you want to draw more than one segment of a circle you can change the start point S with a FOR-NEXT loop and express the end (E) as S plus the desired width of the segment (**Figure 4.17**).

```
45 FOR S=0 TO 1 STEP .2
50 CIRCLE (128,96),R,1,1,S,S+0.0
5
55 NEXT S
```

The first segment will be drawn from 0 to 0.05, the second from 0.2 to 0.25, the third from 0.4 to0.45, the fourth from 0.6to 0.65, the fifth from 0.8 to 0.85.

Figure 4.17 Five segments



In four-colour modes a similar technique which incorporates the colour number into the calculation can be used to produce different coloured segments (**Figure 4.18**). The colour number must be divided down to give a suitable value.

```
20 PMODE 3,1:SCREEN 1,0:PCLS1
40 FOR R=10 TO 90
45 FOR S=0 TO 1 STEP 0.2
46 FOR C=2 TO 4 STEP 2
50 CIRCLE (128,96),R,C,1,S+C/4,S
+C/4+0.1
54 NEXT C
55 NEXT S
60 NEXT R
200 GOTO 200
```

As far as the start and end of the arcs are concerned the innermost loop gives values of 2/4 = 0.5 or 4/4 = 1 and the middle loop steps by 0.2. Pairs of arcs are drawn by the innermost loop. The first arc is drawn in colour 2 from 0.5 to 0.6 and the second in colour 4 from 1 to 1.1 etc., (**Table 4.1**). Note that yellow segments all start from odd numbered points and red segments from even and that if the value is greater than 1 then the 1 has no effect.

Figure 4.18 Coloured segments

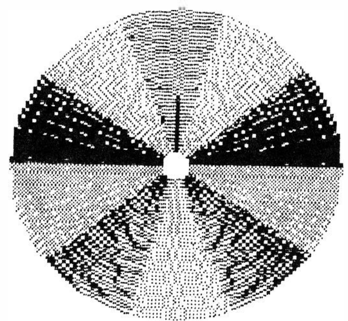


Table 4.1

COLOURING SEGMENTS OF CIRCLES

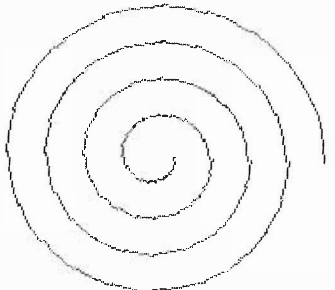
Segment	S	C/4	S+C/4	S+C/4+0.1
1 yellow	0	0.5	0.5	0.6
2 red	0	1.0	1.0	1.1
3 yellow	0.2	0.5	0.7	0.8
4 red	0.2	1.0	1.2	1.3
5 yellow	0.4	0.5	0.9	1.0
6 red	0.4	1.0	1.4	1.5
7 yellow	0.6	0.5	1.1	1.2
8 red	0.6	1.0	1.6	1.7 etc

Spirals

Spirals can also be constructed from small arcs if suitable increments in the diameter are included (**Figure 4.19**).

```
20 PMODE 4,1:SCREEN 1,0:PCLS
40 FOR R=10 TO 90
45 FOR S=0 TO 1 STEP 0.05
50 CIRCLE (128,96),R,1,1,S,S+0.05
54 N=N+1
55 NEXT S
60 NEXT R
200 GOTO 200
```

Figure 4.19 Spiral



A couple of minor changes to the parameters can have major effects on the result (**Figure 4.20**).

```
50 CIRCLE (128,96),R,1,0.4,S,S+0.4
```

Finally a demonstration that judicious juggling with quite simple programs can have quite startling effects. Look at **Figure 4.21** and see how simply it is built up into a 'flying saucer' from a simple ellipse.

```
10 ST=0.1:FI=1.5
20 PMODE 4,1:SCREEN 1,0:PCLS
30 FOR R=20 TO 90 STEP 5
```

```
40 FOR HW=ST TO FI STEP 0.1
50 CIRCLE (128,96),R,1,HW
60 NEXT HW
70 FI=FI-0.3
80 NEXT R
```

Figure 4.20 Elliptical Spiral

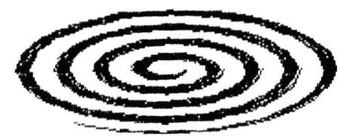


Figure 4.21 Flying saucer

CHAPTER 5
DRAWing

DRAW must be one of the most versatile graphics features available in BASIC, and both its syntax and applications are many and varied. DRAW (like PLAY) always acts on a string and you may find it a little daunting at first as there are no less than fifteen different DRAW commands (**Table 5.1**). However these can be divided up into three main groups according to whether they cause movement, a change in mode, or have other actions.

To be able to see all the DRAW commands in action we must first set up a hi-res screen display. (Enter line 20 blindly for the moment and forget it until later!)

```
10 PMODE4,1:SCREEN1,0:PCLS
20 DRAW"S48"
1000 GOTO 1000
```

Although nothing can be seen on the screen an invisible cursor is now positioned at the screen centre (coordinates 128,96), and this becomes apparent if a DRAW command is now added.

```
30 DRAW"U"
```

A short line will now appear pointing Up from the centre of the screen, and if the string is modified to "URDL" (Up, Right, Down, Left) a small square will be formed (**Figure 5.1**).

```
30 DRAW"URDL"
```

Figure 5.1 Square

Note that each new line is drawn from the point where the last line ended, so that the square is offset towards the top right of the screen. Semicolons between these commands are optional, and are usually left out to save space in the string. If a number follows one of these letters it defines how

Table 5.1

DRAW COMMANDS

MOVEMENT

```
vertical

  U     up     (0 degrees)
  D     down   (180 degrees)

horizontal

  L     left  (270 degrees)
  R     right (360 degrees)

diagonal

  E     at 45 degrees
  F     at 135 degrees
  G     at 225 degrees
  H     at 315 degrees

absolute

  M     draw line to specified
        coordinates
  BM    blank move (move
        to new coordinates
        without drawing)


MODE
----

  A     change angle
  C     change color
  S     change scale

OTHER
-----

  N     no update of last draw
        coordinates
  X     execute a substring
```

many times that particular command is to be repeated, thus U4 will draw a line twice as long as U2 and four times as long as U. If we double the U and D commands in the string we will now produce a vertically-elongated rectangle instead of a square (**Figure 5.2**).

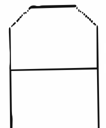
```
30 DRAW"U2RD2L"
```

Figure 5.2 Rectangle

E,F,G and H work in exactly the same way but give the four possible 45 degree diagonal positions. Any combination of these commands can be constructed to form any type of design. For example this string draws a letter A. (**Figure 5.3**).

```
30 DRAW"U5ER2FD5U3L4"
```

Figure 5.3 'Drawn' character



It is not necessary for the string to be actually defined immediately after the word DRAW as this will also act on existing strings.

```
30 A$="U5ER2FD5U3L4"
40 DRAW A$
```

Scale

Now to reveal the secret of line 20which used the very useful scaling feature to effectively multiply the whole of the string by a factor. If no scale is specified then the factor is 1 and U, for example, will draw a line one screen point long. On the other hand, as we called for a scale of 48 in line 20, U actually drew a line 48 points long.

As usual points are set in different ways according to the resolution (PMODE) selected. We have selected PMODE 4 but you should also look at the effect of using lower resolution two and four-colour modes, and remember that the system may not distinguish two coordinates as different points in the lower modes.

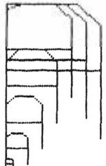
Scale can be altered within a program, provided that you remember that DRAW only acts on strings and not simple variables. To include a simple variable you must first convert it to a string with the STR$ function. To demonstrate this run this routine which will produce a series of boxes which increase in size. **(Figure 5.4).**

```
20 FOR S=4 TO 56 STEP 8
30 DRAW"S"+STR$(S)
40 DRAW"URDL"
50 NEXT S
```

Figure 5.4 Scaled boxes

The maximum value for S is 62 so any further increase in size must be made by adding numbers after the actual motion commands (remembering that these are multiplied by S to produce a cumulative effect). All of our boxes started from the same point, as they also ended at the same point, but if you replace line 40 with the string for the letter R disaster will strike. The first problem is that the drawing does not end at the start position, whilst the second problem is that the letters reach the top of the screen and become distorted **(Figure 5.5)**. These difficulties can be solved by adding 'D3' to the end of the string, so that drawing now ends at the start point, which also incidentally makes enough room for the larger letters.

Figure 5.5 Distortion caused by letter moving up screen each time



The strings which are used for DRAW can be added just like any other strings so lines 30 and 40 could be combined into one:

```
30 DRAW "S"+STR$(S)+"U5ER2FD5U3L
4D3"
```

Colour

So far we have only used a two colour mode but now let's change to PMODE3 to see the operation of the colour command C which works in a similar way to S.

```
10 PMODE3,1:SCREEN1,0:PCLS
20 FOR C=1 TO 4
30 DRAW"C"+STR$(C)+"S24U5ER2FD5U
3L4D3"
40 A$=INKEY$:IF A$="" THEN 40
50 NEXT C
60 GOTO 20
```

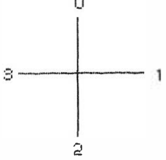
Now each time a key is pressed the letter will be redrawn in a different colour. As colour 1 is the background colour this means that sometimes the letter is erased. The default colour is the current foreground, and once a colour is specified in this way it will be used until it is changed again.

Angle

The final command in the mode group is ANGLE which allows you to change the direction of movement in every following command in 90 degree steps, producing rotation of the design. The steps are defined by the numbers 0 to 3 where 0 is vertical, 1 is 90 degrees, 2 is 180 degrees and 3 is 270 degrees (**Figure 5.6**)

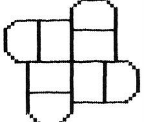
```
20 FOR A=0 TO 3
30 DRAW"A"+STR$(A)+"S24U5ER2FD5U
3L4D3"
50 NEXT A
```

Figure 5.6 Angle numbers



The letter will now be drawn in all four possible directions **(Figure 5.7)**. A point to watch once again is that the defined angle will be used for every subsequent DRAW command until it is changed.

Figure 5.7 Changing angle



No-update

Normally each new DRAW command starts from the last point drawn but it is sometimes useful to be able to draw a number of lines from the same point, so a N (no-update) command is also available. If you put the letter N before any other command then the 'cursor' position will not be changed during that move. This command is applied to each command in this routine to produce a star which radiates from a central point **(Figure 5.8)**

```
20 DRAW"S48NUNENRNFNDNGNLNH"
30 GOTO 30
```

Figure 5.8 Star produced by no-update

It is also useful in producing branching structures when used selectively **(Figure 5.9)**

```
20 DRAW"R20ND15R15ND10R10ND5"
```

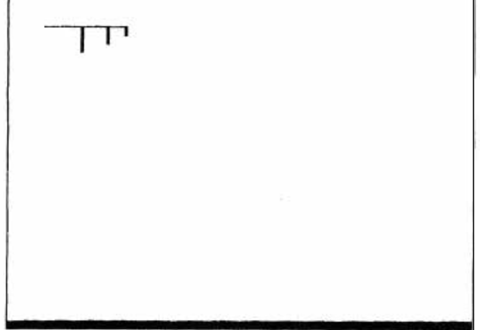
Figure 5.9 Selective use of no-update

Move and Blank Move

In addition to the movement commands discussed so far we also have M (move) and BM (blank move) which are rather different in that they specify actual new coordinates rather than just direction and distance. The only difference between M and BM is that M draws a line to the new coordinates but BM just moves the cursor there without drawing. Until now we have been content to start all our drawing from the default screen centre position, but this can easily be altered by adding BM x,y to the front of our string. The coordinates x and y may be defined in either absolute or relative terms. If numbers alone are included they are taken to be absolute. Thus BM20,20 will move the drawing to the top left of the screen (**Figure 5.10**)

```
20 DRAW"BM20,20R20ND15R15ND10R10
ND5"
```

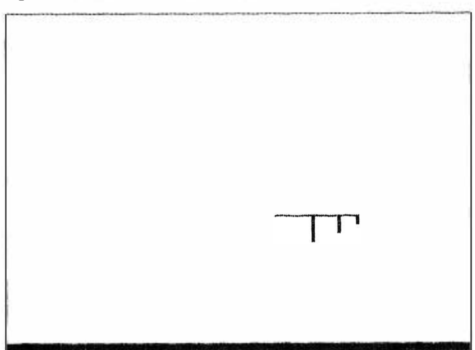
Figure 5.10 Blank move to 20,20



To indicate a relative move you must put a + or − before the number, when the calculation will be made relative to the current position. Now BM+20,+20 will move the drawing towards the bottom right of the screen from the centre (**Figure 5.11**). The rule to remember is that up and left are always negative.

```
20 DRAW"BM+20,+20R20ND15R15ND10R
10ND5"
```

Figure 5.11 Relative blank move to +20, +20



The move command M itself is mainly used to draw relatively long lines to predefined points (rather like LINE). It is possible to put variables in the M and BM commands, although it is a little messy as each variable must be converted independently to a string and these must be separated by a comma. Variable X and Y coordinates are entered here to form a series of lines of differing length **(Figure 5.12)**.

```
20 Y=50
30 FOR X=0 TO 250 STEP 10
40 DRAW "NM"+STR$(X)+","+STR$(Y)
50 Y=Y+2
60 NEXT X
70 GOTO 70
```

Figure 5.12

Execute substring

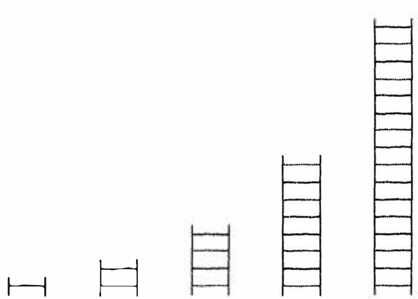
The final command is X which calls a substring which has already been defined elsewhere. The syntax is:

XA$;

and it is important to note that this is the one place where the semicolon is *not* optional (even when it is at the end of a line)! The main value of this command is in complex programs where particular sequences of DRAW commands are to be used frequently. As a simple demonstration we will form a short ladder section as A$ and then build up longer sections by adding strings (but remember the final string length cannot be greater than 255). Each string is then executed to give ladders of varying length, with D$ being executed twice to give the longest ladder **(Figure 5.13)**

```
10 CLEAR 1000:PMODE4,1:SCREEN1,0
:PCLS
20 A$="U10D5R20D5U10BM-20,-0"
30 B$=A$+A$
40 C$=B$+B$
50 D$=C$+C$
60 DRAW"BM10,180XA$;"
70 DRAW"BM60,180XB$;"
80 DRAW"BM110,180XC$;"
90 DRAW"BM160,180XD$;"
100 DRAW"BM210,180XD$;XD$;"
110 GOTO 110
```

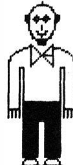
Figure 5.13



CHAPTER 6
Combining the Graphics Commands

A particular design may be built up by a combination of any or all of the graphics drawing commands described so far and to give a final demonstration of how to do this we have enlisted the help of our friend the PIC-MAN (**Figure 6.1**). Perhaps we should explain that he is quite unlike his abbreviated relative PI-MAN in that he is definitely not an automaton and certainly has no political aspirations (hence his appearance in glorious black and white), and that unlike PAC-MAN he has no fear of ghosts or an insatiable appetite for power pills. Instead he has deliberately been constructed from a wide assortment of graphics commands so that he demonstrates how you can combine most of the hi-resolution drawing commands in a single program.

Figure 6.1 PIC-MAN



We start by setting the PMODE to 4 so that we have the highest possible resolution and can therefore add lots of fine detail. SCREEN 1,0 gives us white on a black background.

```
10 PMODE 4,1:SCREEN 1,0:PCLS
```

In its simplest form the CIRCLE command only needs two parameters: the X and Y screen coordinates of the centre, and the diameter of the circle, so that will do nicely for a pair of small round eyes (**Figure 6.2**). Remember that coordinates are always specified on a 256 x 192 grid, no matter which PMODE you are using. When planning a design you can use graph paper

or fancy plotting sheets, but a trial and error approach on the screen is often quicker where there is a lot of fine detail to squeeze in. There is no need to specify anything else as the default values will give you a full circle in the foreground colour.

```
40 CIRCLE(79,48),2
50 CIRCLE(84,48),2
```

Figure 6.2 Eyes

Heads are not actually round but rather egg-shaped (especially if you are a micro-maniac) so for that we need to form a vertically-distorted ellipse with CIRCLE. It is the height/width (HW) ratio which allows you to include this distortion but note that this must be the FOURTH parameter. It is very easy to forget that the system can only tell that this is the fourth item if it can see three other parameters before this, and that therefore you must now also include the third parameter (colour). Although we have actually put the number I in to set the colour to white the computer will also recognise a comma on its own as the default value, so either of the following lines has the same effect. In this program we have deliberately included all the actual values to make it easier to read. The HW ratio is greater than 1 so that distortion is vertical rather than horizontal (**Figure 6.3**)

```
30 CIRCLE(82,50),8,1,1.5
```

or

```
30 CIRCLE(82,50),8,,1.5
```

Figure 6.3 Head

A further feature of CIRCLE is the ability to form only certain arcs of the whole circle, using parameters five and six to set the start and finish. PIC-MAN is smiling so his mouth is the bottom half of a circle which is only drawn from 0 (3 o'clock) to 0.5 (9 o'clock) (**Figure 6.4**).

```
70 CIRCLE(82,55),3,1,1,0,.5
```

Figure 6.4 Mouth

The simplest sort of LINE just goes from one point to another, as in the nose, and PSET rather than PRESET means that white (the foreground colour) is used (**Figure 6.5**).

```
60 LINE(82,52)-(82,54),PSET
```

Figure 6.5 Nose

Although his ears may look positively princely they are rather too small to form with CIRCLE so are simply boxes formed by specifying the top left and bottom right corners and adding B to the end of the LINE command. The neck is made the same way (**Figure 6.6**).

```
80 LINE(73,46)-(74,49),PSET,B
90 LINE(90,46)-(91,49),PSET,B

100 LINE(80,61)-(84,63),PSET,B
```

Figure 6.6 Ears and neck

Now that we have a neck we can add the round-shouldered look by a combination of all the previous CIRCLE ideas to give the top half of a horizontally-distorted ellipse (**Figure 6.7**)

```
110 CIRCLE(82,72),18,1,.5,.5,1
```

Figure 6.7 Shoulders

We could have continued to use LINE to draw the rest of his body but DRAW is more versatile as a whole series of lines in different directions can be DRAWn at the same time. First we make the top half of the body (**Figure 6.8**).

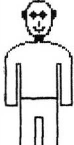
```
120 DRAW"BM64,72D28R6U24R2D20R26
U20R2D24R6U28"
```

Figure 6.8 Arms and trunk

and then the bottom (**Figure 6.9**).

```
130 DRAW"BM72,97D30R8U20R4D20R8U
30"
```

Figure 6.9 Legs



Note the use of blank moves (BM) to set the starting position, and make sure you follow the instructions round to see which way they go. It is best to try to plan your route carefully so that it is as compact as possible. You must also always remember that the next DRAW command will normally start from the last point DRAWn, even if that was done an hour or more ago (as long as you don't use RUN). So if things start going haywire in your programs look back and check what was the last thing DRAWn!

DRAW can be used to make any sort of design and another place where it is very useful is in putting text on the high resolution screen. The letters forming the title PIC-MAN are DRAWn in this way. (For more details of this technique see later) (**Figure 6.10**).

```
20 DRAW"BM150,100S8U6R3FDGL3BM+8
,+3R2LU6LR2BM+5,+6HU4ER2FHL2GD4F
R2EBM+2,-2R4BM+2,+3U6F2E2D6BM+4,
+0U5ER2FD5U3L4BM+8,+3U6DF4DU6S4"
```

Figure 6.10 Lettering

PIC-MAN obviously favours Doc Marten's as his boots are quite massive top halves of CIRCLEs with thick soles formed by boxes FILLED with the foreground colour (**Figure 6.11**).

```
140 CIRCLE(76,132),5,1,1,.5,1
150 CIRCLE(88,132),5,1,1,.5,1
160 LINE(71,132)-(81,134),PSET,B
F
170 LINE(83,132)-(93,134),PSET,B
F
```

Figure 6.11 Boots

To make him look more solid we have PAINTed in his trousers (**Figure 6.12**). PAINT will fill an area with the first specified colour until it reaches the second specified colour and the main user difficulties are making sure you set the right coordinates and that there are no holes through which PAINT can leak. Try altering the coordinates in line 180 and watch what happens.

```
180 PAINT(74,100),1,1
```

Figure 6.12 Painted trousers

DRAW always acts on a string but this string can also be defined in advance as a substring and used repeatedly by means of the X command. As we have two identical hands to DRAW these have first been defined as H$. H$ also uses the useful no-update or N parameter. Normally each new DRAW command continues from where the last line DRAWn ended, but if you put N in front of a command then the next line is DRAWn from the same place as the current one. Follow the sequence carefully to see how each finger is formed (his thumbs are out of sight in case you think he is deformed).

```
190 H$="ND5R2ND5R2ND5R2ND5"
```

To put the hands into the appropriate positions we just need to set the new screen start position and then execute H$ by sandwiching it between 'X' and ';' (**Figure 6.13**).

```
200 DRAW"BM64,100XH$;"
210 DRAW"BM94,100XH$;"
```

We are afraid that PIC-MAN is really rather pompous and has taken to wearing the bow-tie defined in A$. Notice that this is DRAWn from the centre using some of the diagonal commands (F and G) and that it is delib-

erately asymmetrical. A relative blank move is used to separate the final short stripe from the rest of the picture (**Figure 6.14**). This has the advantage that you do not have to calculate the actual position, but only the displacement from the current position as + and − a number of screen points. It is not usually essential to start DRAWing a design from the centre but in this case PIC-MAN wants to prove to you that this is actually a revolving bow-tie, which grows, so he needs a central point to work from!

```
220 A$="BM82,69F3U6G6U6F3BM+1,+0
R2":S=4
```

Figure 6.13 Hands

Figure 6.14 Complete

The scale parameter S sets the size of the string DRAWn, the angle parameter A allows you to change the direction of DRAWing by 90 degree steps, and the colour parameter C allows you to change the colour of DRAWing. You can use a variable to change any of these provided that you first convert the variable to a string with STR$. All these ideas have been combined together in this little sequence in which the tie is DRAWn in colour 1 and then colour 0 (ie drawn and erased), in all possible directions,

and at ten different increasing scales. The sound is included to slow things down so that the movement can be clearly seen **(Figure 6.15)**

```
230 DRAW"S"+STR$(S)
240 FOR N=0 TO 3:DRAW"A"+STR$(N)
250 FOR M=1 TO 0 STEP-1:DRAW"C"+
STR$(M)+A$
260 SOUND255,1:NEXT M,N
270 S=S+1:IF S<10 THEN 230
```

Figure 6.15 Tie revolving

PIC-MAN

Of course pride always comes before a fall and that rotating tie looks very dangerous, so it is hardly surprising that it eventually explodes. Explosions are very frequent features of computer programs so this is a very general routine. A series of expanding concentric CIRCLEs are drawn by using the variable X to set the diameter, and a sound is integrated wth each expansion of the circle **(Figure 6.16)**. PLAY is used instead of SOUND as it allows the use of a much shorter duration if tempo (T) and note-length (L) are set to their highest value (255).

```
280 FOR X=1 TO 59 STEP 2:CIRCLE(
82,69),X,1:PLAY"T255L255CD":NEXT
 X
```

Once it has passed its peak the explosion dies away as the CIRCLEs are now drawn in reverse order by STEP-1, so that only PIC-MAN's boots and a few fragments remain **(Figure 6.17)**. Notice that integration of graphics and sound is more complete here as X also varies the volume and tempo of the PLAY command.

```
290 FOR X=59 TO 1 STEP-1:CIRCLE(
82,69),X,0:PLAY"L255V"+STR$(INT(
X/2))+"T"+STR$(X*4)+"DC":NEXT X
```

Figure 6.16 Explosion

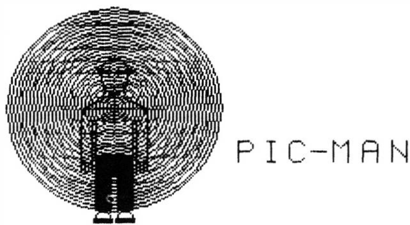


Figure 6.17 Remains

CHAPTER 7
On-Screen Movement

Some type of on-screen movement is a very common requirement in graphics programs, but it can take many different forms. The simplest type of movement deals only with a single point which is moved from one position to another.

Text screen

The PRINT positions on the text screen are mapped as 16 lines of 32 characters numbered sequentially from 0 to 511, and this is taken account of in this routine which will move a black block (CHR$(128)) in any of four directions.

```
10 CLS:P=240
20 A$=CHR$(128)
30 CLS
40 PRINT @ P,A$;
60 I$=INKEY$:IF I$="" THEN 60 EL
SE I=ASC(I$)
70 IF I=8 OR I=21 THEN P=P-1
80 IF I=9 OR I=93 THEN P=P+1
90 IF I=94 OR I=95 THEN P=P-32
100 IF I=10 OR I=91 THEN P=P+32
110 IF P<0 THEN P=0 ELSE IF P>51
0 THEN P=510
150 GOTO 30
```

Note that the ASCII codes for both normal and shifted cursor characters are tested for, and that the limit checks in line 110 prevent you crashing if you try to leave the screen. The last screen position (511) is not used as PRINTing here causes automatic screen scroll.

As the routine returns to the CLS in line 30each time, the character in the old print position is automatically erased. If you want to be more selective and only erase the character in the last position you need to store the old position as a new variable (LP) and then reset this point as you move with a PRINT @LP,B$. A$ and B$ can be set to any of the alphanumeric or

graphic characters. In the example below (CHR$(143)) is used to reset the old point to green.

```
10 CLS:P=240:LP=P
20 A$=CHR$(128):B$=CHR$(143)
30 PRINT @ LP,B$
50 LP=P
```

Non-destructive movement

In more complex programs you may wish to move without permanently altering the screen and you must then keep a record of what lies in the new print position. You can PEEK into the new position but unfortunately the PEEK values are not always the same as the ASCII codes. You must therefore either use a sorting routine to calculate the appropriate ASCII code or store the PEEK value as a simple variable and then POKE this back instead of PRINTing a string. (The text screen starts at location 1024, so that POKE(1024+X) is the same as PRINT @ X). So that only one variable is needed you must reset the old position before you PEEK the new one. Note that line 30 must now be deleted and that a message has been added to line 10 so that you see the replacement of the display working. PE must also be initially set to the value PEEKed in the start position.

```
10 CLS:PRINT @ 256,"THIS IS A TE
ST":P=240:LP=P:PE=PEEK(1024+P)
20 A$=CHR$(128)
30 (deleted)
130 POKE(1024+LP),PE
140 PE=PEEK(1024+P)
150 GOTO 40
```

Text cursor

Where this type of routine is used to move a cursor over text it is usual to invert the screen display of the character at the cursor position, so that the character is still visible. As the ASCII codes for lower case (inverse video) characters are all 32 greater than those for the appropriate upper case (normal) characters we only need to add 32 to the PEEKed value.

```
40 PRINT @ P,CHR$(PE+32);
```

Graphics

Essentially similar routines can be used to deal with individual pixels in both low and high-resolution graphics. In both cases calculating the moves is even simpler as the screen is mapped here as X and Y coordinates. In low-res the screen is 64 (0–63) by 32 (0–31), SET will turn a pixel to any colour (Cl) and RESET will revert to the background colour (black).

```
10 CLS0:X1=32:Y1=16:X2=X1:Y2=Y1
20 C1=4
30 RESET(X2,Y2)
40 SET(X1,Y1,C1)
50 X2=X1:Y2=Y1
60 I$=INKEY$:IF I$="" THEN 60 EL
SE I=ASC(I$)
70 IF I=8 OR I=21 THEN X1=X1-1
80 IF I=9 OR I=93 THEN X1=X1+1
90 IF I=94 OR I=95 THEN Y1=Y1-1
100 IF I=10 OR I=91 THEN Y1=Y1+1
110 IF X1<0 THEN X1=0 ELSE IF X1
 >63 THEN X1=63
120 IF Y1<0 THEN Y1=0 ELSE IF Y1
 >31 THEN Y1=31
150 GOTO 30
```

The hi-res equivalents PSET and PRESET operate in the same way as SET and RESET but on a 256 by 192 grid.

```
10 PMODE 3,1:SCREEN 1,0:PCLS:X1=
128:Y1=96:X2=X1:Y2=Y1
30 PRESET(X2,Y2)
40 PSET(X1,Y1,C1)
110 IF X1<0 THEN X1=0 ELSE IF X1
 >255 THEN X1=64
120 IF Y1<0 THEN Y1=0 ELSE IF Y1
 >192 THEN Y1=32
150 GOTO 30
```

Unfortunately, although POINT can be used to test the colour of an individual low-resolution pixel, this value cannot be incorporated into a SET command, so that the original display can be recreated after movement, as 0 (black) is not a valid colour. On the other hand the high-resolution equivalent PPOINT value will be a colour which can be used in PSET to recreate the previous state of the cursor point.

```
20 C1=4:C2=PPOINT(X1,Y1)
30 (deleted)
130 PSET(X2,Y2,C2)
140 C2=PPOINT(X1,Y1)
150 GOTO 40
```

The cursor will now move non-destructively over the screen, even if the background colour changes. This can be checked by changing the PCLS in

line 10 to a different colour or, more dramatically, by adding this line to give a coloured design to move over.

```
15 CIRCLE(128,96),10,2:PAINT(128
,96),2,2:CIRCLE(128,96),20,3
```

The only problem now is that the red cursor is invisible when it is on a red background. This can be solved by a check that the cursor colour (C1) is different to the background colour (C2), followed by a change of cursor colour if appropriate.

```
140 C2=PPOINT(X1,Y1):IF C2<>C1 T
HEN 40 ELSE IF C1<3 THEN C1=C1+2
 ELSE C1=C1-2
```

Flashing cursor

On a complex screen it may be difficult to see the cursor, so this is often turned on and off to give a flashing effect. This is simply achieved by modifying the key-check line so that if no key is pressed the cursor is erased and then redrawn. For example:

```
60 I$=INKEY$:IF I$="" THEN PRESE
T(X1,Y1):GOTO 40 ELSE I=ASC(I$)
```

The rate of flashing can be slowed by inserting a timing loop.

```
60 I$=INKEY$:IF I$="" THEN PRESE
T(X1,Y1):FOR T=1 TO 10:NEXT T:GO
TO 40 ELSE I=ASC(I$)
```

Controlling movement

If you want to travel faster you can move more than one pixel at each decision (increment X1 and Y1 by more than 1), although this may or may not give less precise control. Remember that in PMODE 4 each coordinate refers to a single screen point, but in lower PMODEs points are set in groups so that PSETting more than one coordinate can have the same effect. For example in PMODE 0 you must increment both X1 and Y1 in steps of at least 2, and in PMODE 3 you should increment X1 in steps twice the size of those used for Y1.

Using INKEY$ for continuous movement can be tedious, as the ROM keyboard debounce routine requires you repeatedly to lift your finger and release the key. However this problem can be got around easily by using this subroutine where you would normally use INKEY$. It waits for a key to be pressed and then autorepeats until the key is released. PEEK(135) gives the ASCII code for the key pressed.

```
60 ON (PEEK(337)<255)+1 GOTO 60
65 I=PEEK(135)
```

Where joysticks are used to directly control screen position appropriate scaling of the JOYSTK values is required. For example in low-res the JOYSTK(1) value for the Y coordinate must be halved to give 0–31.

```
1000 X1=JOYSTK(0):Y1=JOYSTK(1)/2
```

On the other hand in PMODE 4 JOYSTK(0) must be multiplied by 4 (to give 0–255) and JOYSTK(1) by 3 to give (0–191).

```
1000 X1=JOYSTK(0)*4:Y1=JOYSTK(1)
*3
```

Although this scaling for hi-res gives numbers in the appropriate ranges you must remember that not all the individual points can now be reached, as movement is being made in steps of 4 and 3.

It is also possible to use a joystick to control direction of movement rather than absolute position and in this case all points are easily reached. The actual joystick values are read into temporary variables J0 and J1 which are used to determine direction.

```
1000 J0=JOYSTK(0):J1=JOYSTK(1)
1020 IF J0=<20 THEN X1=X1-1 ELSE
IF J0=>50 THEN X1=X1+1
1030 IF J1=<20 THEN Y1=Y1-1 ELSE
IF J1=>50 THEN Y1=Y1+1
```

When the lever is central both J0 and J1 will equal 32, so upper and lower limits of 20 and 50 have been arbitrarily chosen, but the sensitivity (distance lever must be moved before an effect is produced) can be altered by varing these limit values. Values close to 32 give most rapid response but it can then be difficult to prevent unwanted movement. As X and Y coordinates are considered separately diagonal movements can be produced. If a wait status is needed a check for a central position on both axes can be included.

```
1010 IF J0>20 AND J0<50 AND J1>2
0 AND J1<50 THEN 1000
```

Joystick position can also be used to control rate of movement if the size of step is linked to the distance of the lever from the central position. In this routine three different size steps are provided (1, 2 and 5 units). The most extreme positions must always be checked for first.

```
1020 IF J0<10 THEN X1=X1-5 ELSE
IF J0<15 THEN X1=X1-2 ELSE IF J0
 <20 THEN X1=X1-3 ELSE IF J0>60
THEN X1=X1+5 ELSE IF J0>50 THEN
X1=X1+2 ELSE IF J0>40 THEN X1=X1
+1
```

Moving more than one point

So far we have only looked at the simplest situation where a single point is being moved, but often we need to move a series of linked points which make up a particular design. We will consider the position for low-resolution first.

Direct setting of points

The simplest way to form your chosen design is to SET the required colour at each desired screen point, which is defined as X and Y coordinates. Unless the number of points to be SET is very small these coordinates are normally stored in DATA statements. This DATA can then be READ and the corresponding points set.

```
30 CLS0
40 C1=1
50 FOR N=1 TO 16:READ X,Y:SET(X,
Y,C1):NEXT N
80 DATA 0,0,0,1,0,2,0,3,1,0,1,1,
1,2,1,3,2,0,2,1,2,2,2,3,3,0,3,1,
3,2,3,3,4,0,4,1,4,2,4,3
```

Note that the screen must be cleared to black (CLS0) and if the DATA is to be used more than once a RESTORE must be added to the end of line 50.

```
20 FOR R=1 TO 100
50 FOR N=1 TO 16:READ X,Y:SET(X,
Y,C1):NEXT N:RESTORE
60 NEXT R
```

The display will now flash as the points will be SET and the screen cleared 100 times.

Reading into arrays

A modification of this approach is to READ the DATA into X and Y ARRAYS and then use the array elements in the SET command.

```
1 GOSUB 90
50 FOR N=1 TO 16:SET(X(N),Y(N),C
1):NEXT N
90 DIM X(16),Y(16)
100 FOR N=1 TO 16:READ X(N),Y(N)
:NEXT N:RETURN
```

At first sight this seems rather complicated, but this method has two main advantages. The first gain is a small decrease in the time taken to plot the design on the screen. This can be demonstrated by timing a 100 loop cycle.

```
10 TIMER=0
70 PRINT TIMER/50
```

Comparative timer values are 17.8 seconds for direct READing and 16.8 seconds using the array method.

One problem with DATA statements is that they can only be read sequentially and the second, more important, advantage of transferring the values to an array is that any part of the design can easily be used independently. This procedure is dealt with in more detail later, but the general idea can be seen by adding a STEP 2 in line 50 so that only alternate points are SET.

```
50 FOR N=1 TO 16 STEP 2:SET(X(N)
,Y(N),C1):NEXT N
```

The 'characters' approach

As you should remember from the earlier explanation of low-res graphics, there are restrictions on the way points can be SET to different colours and any pattern of set points can also be represented by certain of the Dragon graphics characters. Although using this method requires some careful planning with a jigsaw approach on squared paper to determine the appropriate characters it has a number of plus points. Each graphics character is equivalent to four pixels so that the number of values to be entered is divided by 4. The graphics characters can be PRINTed, so that there is no need for a long, complicated, and often error-prone, DATA statement. 'Impossible' combinations of points will be obvious and there is also a large increase in speed of plotting. The simplest action is to put the characters directly into the PRINT command.

```
50 FOR N=0 TO 1:PRINT @ 32*N,CHR
$(143);CHR$(143);:NEXT N
```

This new line 50 produces the same display as the old line, by means of a loop which PRINTs two characters on adjacent lines. However it is much quicker (5.1 seconds) — only about 30% of the time taken to SET each point individually.

The speed can be increased even more if the design is stored as a string variable.

```
1 A$=CHR$(143)+CHR$(143)
50 FOR N=0 TO 1:PRINT @ 32*N,A$;
:NEXT N
```

This minor change has profound effects (3.2 secs) as the time drops to only 18% of the original. If you double the number of points to be SET you will virtually double the time taken (34.1 seconds for direct READing or 30.2 seconds from an array). On the other hand increasing the length of A$ to 10 or even 20 characters has much less effect (3.7 seconds and 4.2 seconds, respectively).

Where movement of the design is required rapid replotting is usually necessary, and the logic of the 'characters' rather than the 'SET' approach should now be obvious! If you require even more speed you can POKE &HFFD7,0 (if your machine can stand this) which will reduce all times by a further 25%.

Referenced movement

The easiest way to define the screen position of the design is by means of offsets from the start position. For X and Y mapping we will call these XO (for X axis offset) and YO (for Y axis offset). To determine the new coordinates we simply add or subtract XO and YO from the current position as described earlier in the general movement section, and then add these offsets to the SETting line.

```
50 FOR N=1 TO 16:SET(X(N)+XO,Y(N
)+YO,C1):NEXT N
```

Where graphics characters are used we are back to the sequential 0–511 arrangement so we must allow for steps of 32 on the Y axis, either in the key-check line or in the PRINT @ line.

Modifying limits

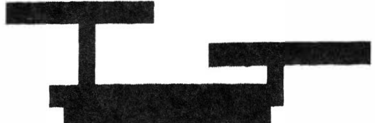
Where more than one screen position is to be set at the same time we must modify our limits to ensure that there is room for the whole of the pattern. If we use the top left-hand corner of our area as the reference point we will need to subtract the width of the design from the X axis limit and the depth from the Y axis limit. For example a 4 pixel by 4 pixel design on the low-res screen cannot be moved beyond X,Y coordinates of (63−4),(31−4). If you look back to the cursor routine you will see that as a general procedure the size could be defined as XS and YS and then these values subtracted from the absolute limits.

```
10 CLS0:X1=32:Y1=16:X2=X1:Y2=Y1:
XS=4:YS=4
110 IF X1<0 THEN X1=0 ELSE IF X1
 >(63-XS) THEN X1=(63-XS)
120 IF Y1<0 THEN Y1=0 ELSE IF Y1
 >(31-YS) THEN Y1=(31-YS)
```

Starship

Let's put all these principles of moving a low-res design together in a program which moves a more complicated picture of a starship (**Figure 7.1**) around the screen, instead of an anonymous box.

Figure 7.1 Starship



The first idea is to map out the design on the low-res grid, put all the pixel X and Y coordinates into DATA statements, read these into arrays, and index the SET coordinates to cursor key movement. The transfer of DATA to the arrays is put in a subroutine at the end of the program out of the way. On return from this initialisation process all the points are SET by line 30 and the program waits for a key press. The cursor key routine used here PEEKs at certain reserved memory locations to detect which key is pressed and has the advantage over INKEY$ of autorepeating.

As soon as a key is pressed (PEEK(337)< 255) each point is RESET in turn in line 50. Note that this must be done BEFORE the screen position is updated.

When specifying limits you must not forget to allow for the actual width and height of the picture. If you check the DATA the lowest X co-ordinate is 5 and the highest 29, and similarly the limits of the Y coordinates are 1 and 6. The overall size of the picture is therefore 25 by 6 pixels. The design is SET from the top left point which starts at (5,1), rather than the more obvious start coordinates of 0,0, so the right-hand X limit must be (X axis length (64) — picture width (25) — original distance from left (5)) = 34, the bottom Y limit (Y axis length (32) — picture height (6) — original distance from top (1)) = 25, and the left-hand X and upper Y limits are (0 — original distance from left (5)) = −5 and (0 — original distance from top (1)) = −1, respectively.

```
10 GOSUB 1000
20 CLS0
30 FOR N=1 TO 55:SET(X(N)+XO,Y(N
)+YO,C):NEXT N
40 IF PEEK(337)=255 THEN 40
50 FOR N=1 TO 55:RESET(X(N)+XO,Y
(N)+YO):NEXT N
60 IF PEEK(341)=223 THEN YO=YO-1
70 IF PEEK(342)=223 THEN YO=YO+1
80 IF PEEK(343)=223 THEN XO=XO-1
90 IF PEEK(344)=223 THEN XO=XO+1
100 IF XO<5 THEN XO=5 ELSE IF XO
>34 THEN XO=34
110 IF YO<1 THEN YO=1 ELSE IF YO
>25 THEN YO=25
120 GOTO 30
1000 DIM X(55),Y(55)
1010 FOR N=1 TO 55:READ X(N),Y(N
):NEXT N:RETURN
5000 DATA 5,1,6,1,7,1,8,1,9,1,10
,1,11,1,12,1,13,1,14,1,10,2,10,3
,10,4,8,5,9,5,10,5,11,5,12,5,13,
5,14,5,15,5,16,5,17,5,18,5,19,5,
20,5,21,5,22,5,23,5,9,6,10,6,11,
6,12,6,13,6,14,6,15,6,16,6,17,6,
18,6,19,6,20,6,21,6,22,6
5010 DATA 23,4,19,3,20,3,21,3,22
,3,23,3,24,3,25,3,26,3,27,3,28,3
,29,3
```

When you RUN this routine you will find it functions but that it is very slow, taking about 1.2 seconds for each position update. It looks as if this

starship is just drifting in space as it takes more than 45 seconds to cross the screen. Obviously something must be done to speed things up, so why not replace all that RESETting of the SET points in line 50 by a simple CLS0. Not surprisingly that effectively halves the time per update to 0.6 seconds as only half the work is left. Although CLS0 will delete any text you have on the screen remember that both CLS and PRINT are almost instantaneous, so that it is quicker to wipe the whole screen and PRINT back any text you need rather than use RESET.

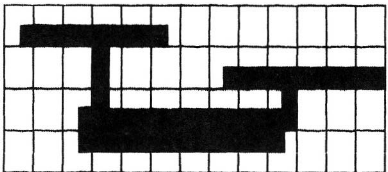
As this stands the screen is cleared before all the key checks and calculations, but there would be less flicker if we deleted line 50 and jumped back to the CLS0in line 20so that the original display was maintained until the new one was about to be plotted.

```
120 GOTO 20
```

To try to speed things up even further we could use PRINT CHR$ instead of SET. A copy of the starship design transferred to the PRINT @ grid is shown in **Figure 7.2**. The individual PRINT @ positions and character codes could be treated as DATA and put into arrays just as for the SET coordinates, and then these printed from the array elements with an offset. Make sure you include a semi-colon after the PRINT!

```
30 FOR N=1 TO 29:PRINT @ X(N)+X0
,CHR$(Y(N));:NEXT N
1010 FOR N=1 TO 29:READ X(N),Y(N
):NEXT N:RETURN
1020 DATA 2,129,3,131,4,131,5,13
1,6,131,7,130,37,138,41,129,42,1
31,43,131,44,131,45,131,46,131,6
8,131,69,139,70,131,71,131,72,13
1,73,131,74,131,75,135,100,132,1
01,140,102,140,103,140,104,140,1
05,140,106,140,107,136
```

Figure 7.2 Starship formed with characters



Although this doubles the speed again (0.3 seconds) it still only looks like impulse drive as a lot of time is spent recovering the array values during each update. If you think the speed of movement is more than doubled remember that each move now takes you twice as far (one character is two pixels high and two pixels wide). For really high speed warp drive operation we are going to have to get rid of those arrays and print strings directly.

The design is four character units high so we need four strings (A$–D$). These are defined in lines 1000–1030, and printed in the correct position relative to each other by appropriate numbers for the start position in the PRINT @ commands (0, 35, 66, 98). Notice that B$ has three black characters (128) included as this is simpler than splitting this string into two.

```
30 PRINT @ 0+PO,A$;:PRINT @ 35+P
O,B$;:PRINT @ 66+PO,C$;:PRINT @
98+PO,D$;
1000 A$=CHR$(129)+STRING$(4,131)
+CHR$(130)
1010 B$=CHR$(138)+STRING$(3,128)
+CHR$(129)+STRING$(5,131)
1020 C$=CHR$(131)+CHR$(139)+STRI
NGS$(5,131)+CHR$(135)
1030 D$=CHR$(132)+STRING$(6,140)
+CHR$(136)
1040 RETURN
```

You will find that the update time is now reduced very dramatically to 0.04 seconds, that is 30 times faster than our original program. The cursor key routine could be altered so that there is only one offset (PO) and this must then be changed by steps of 32 for vertical movement. If you fit the ship in the bottom right-hand corner of the screen you will see that the limiting PRINT @ position is 403.

```
60 IF PEEK(341)=223 THEN PO=PO-3
2
70 IF PEEK(342)=223 THEN PO=PO+3
2
80 IF PEEK(343)=223 THEN PO=PO-1
90 IF PEEK(344)=223 THEN PO=PO+1
100 IF PO<0 THEN PO=0 ELSE IF PO
>403 THEN PO=403
110 GOTO 30
```

The only problem remaining now is that the ship wraps round if it reaches the edge of the screen, because the PRINT @ positions are mapped

sequentially rather than as X,Y coordinates. To solve that we need to analyse the cursor movements in terms of line (L) and row (R) rather than simply PRINT @ position, and calculate PO as (L*32)+R. There is no need to leave the check for absolute PRINT position as a value above 403 cannot now be reached as R would be greater than 19 and L greater than 11.

```
60 IF PEEK(341)=223 THEN L=L-1
70 IF PEEK(342)=223 THEN L=L+1
75 IF L<0 THEN L=0 ELSE IF L>1
1 THEN L=11
80 IF PEEK(343)=223 THEN R=R-1
90 IF PEEK(344)=223 THEN R=R+1
95 IF R<0 THEN R=0 ELSE IF R>1
9 THEN R=19
100 PO=(L*32)+R
110 GOTO 30
```

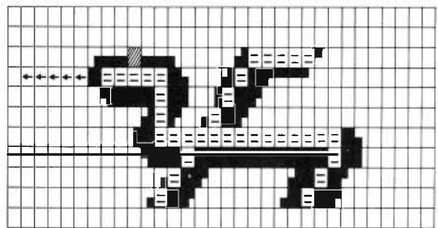
An easier way to deal with CHR$

As an example of a 'consequences' routine in *Dragon Games Master* we gave a design for a Dragon which used a lot of low-res graphics characters **(Figure 7.3)** but required that you typed in the following lines to form the figure.

```
4000 CLS0:PRINT @ 41,CHR$(241);C
HR$(243);CHR$(243);CHR$(159);CHR
$(243);CHR$(243);CHR$(242);STRIN
G$(4,128);CHR$(247);STRING$(5,61
);CHR$(248);
4010 PRINT @ 73,CHR$(255);STRING
$(5,61);CHR$(255);CHR$(242);STRI
NG$(2,128);CHR$(247);CHR$(61);CH
R$(255);CHR$(254);STRING$(2,252)
;CHR$(248);
4020 PRINT @ 106,CHR$(253);STRIN
G$(3,255);CHR$(61);CHR$(255);CHR
$(250);CHR$(128);CHR$(245);CHR$(
61);CHR$(255);CHR$(254);
4030 PRINT @ 141,CHR$(247);CHR$(
61);CHR$(254);CHR$(128);CHR$(241
);CHR$(61);CHR$(255);CHR$(254);
4040 PRINT @ 172,CHR$(245);CHR$(
255);STRING$(14,61);CHR$(255);CH
R$(242);
4050 PRINT @ 205,CHR$(253);STRIN
```

```
G$(2,255);CHR$(61);STRING$(10,25
5);CHR$(61);CHR$(255);CHR$(250);
4060 PRINT @ 233,CHR$(241);CHR$(
61);CHR$(255);CHR$(254);STRING$(
6,128);CHR$(247);CHR$(255);CHR$(
61);CHR$(255);CHR$(248);
4070 PRINT @ 269,CHR$(241);CHR$(
61);CHR$(255);CHR$(254);STRING$(
6,128);CHR$(241);CHR$(255);CHR$(
61);CHR$(255);CHR$(254);
```

Figure 7.3 Dragon



As you can see that was a bit tedious, very vulnerable to typing mistakes, and rather difficult to edit, so we put our minds to finding an easier way of dealing with long strings of characters.

If you look through the lines above you will see that a total of only 14 different characters are used, so we could store their codes as DATA and then READ those numbers into an array.

```
10 DATA 241,243,159,242,128,247,
61,248,255,254,252,250,245,253
20 CLEAR 1000:DIM A(14):CLS0
30 FOR CH=1 TO 14:READ A(CH):NEX
T CH
```

Each element in the array describes one of the characters **(Figure 7.4)**, and we can now describe each line of the figure by a string of numbers instead of having to repeatedly type CHR$(. . .), provided that we have a decoding routine which can convert our number pattern into CHR$. For example the first line would now be:

```
110 A$=" 1 2 2 3 2 2 4 5 5 5 5 6
 7 7 7 7 7 8":GOSUB 1000
```

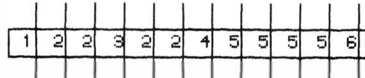
Figure 7.4 Character representation in array

CHR$	241	243	159	242	128	...
ARRAY	1	2	3	4	5	

As we have numbers greater than 9 we must enter the low numbers with a leading space so that we can always slice the string into blocks of two (lines 1000–1010) (**Figure 7.5**) and then take the VAL of the resulting string (B$), thus converting it into an actual number. Line 1030 then looks in the array A and prints the character corresponding to the number in the array element with that subscript. When all of the string has been decoded the PRINT position is moved to the next line and a RETURN made.

```
1000 FOR C=1 TO LEN(A$) STEP 2
1010 B$=MID$(A$,C,2)
1020 B=VAL(B$)
1030 PRINT CHR$(A(B));
1050 NEXT C
1060 PRINT:RETURN
```

Figure 7.5 Slicing the string



Lines 120–180 describe the rest of the figure and if you RUN this you will see the Dragon created, but rather slowly (3.7 seconds). Line 300 calls a subroutine which halts the program until a key is pressed (1070). To make life simpler at a later stage all the numbers in each A$ start from the same screen row, even where this means leading blanks must be inserted.

```
120 A$=" 9 7 7 7 7 7 9 4 5 5 6 7
 9101111 8":GOSUB 1000
130 A$=" 514 9 9 9 7 912 513 7 9
10":GOSUB 1000
140 A$=" 5 5 5 5 6 710 5 1 7 910
```

```
":GOSUB 1000
150 A$=" 5 5 513 9 7 7 7 7 7 7 7
 7 7 7 7 7 7 7 9 4":GOSUB 1000
160 A$=" 5 5 5 514 9 9 7 9 9 9 9
 9 9 9 9 9 9 7 912":GOSUB 1000
170 A$=" 5 5 5 5 5 1 7 910 5 5 5
 5 5 5 6 9 7 9 8":GOSUB 1000
180 A$=" 5 5 5 5 1 7 910 5 5 5 5
 5 5 1 9 7 910":GOSUB 1000

300 GOSUB 1070
1070 IF PEEK(337)=255 THEN 1070
ELSE RETURN
```

If you make a mistake in entering each line it is much simpler to correct, but that sort of speed of operation is not much use. The next stage is to make another array (L$) to hold an image of each line of the design once it is formed. Each new character is added onto the end of L$(n) (line 1040), and L is incremented to move to the next element of L$ to store each new line.

```
20 CLEAR 1000:DIM A(14):DIM L$(8
):CLS0
1040 L$(L)=L$(L)+CHR$(A(B))
1060 L=L+1:PRINT:B$="":RETURN
```

We can now recreate the figure from the L$ array very simply and much quicker than before. A check reveals that it now takes 0.08 seconds which is only 2% of the earlier figure!

```
310 CLS0
320 FOR L=0 TO 7:PRINT L$(L):NEX
T L
400 GOSUB 1070
```

If we use a PRINT @ command related to the line number (P*32), rather than just a PRINT, we can follow our string with a semi-colon and preserve the black screen. (This was why we made sure we started all the lines in the same screen row).

```
410 CLS0
420 FOR P=0 TO 7:PRINT @(P*32),L
$(P);:NEXT P
500 GOSUB 1070
```

As it stands we are PRINTing the figure in the top-left corner but we could easily offset this, by adding another variable (PO). A value of 200 sets the figure in the bottom right of the screen.

```
510 CLS0
520 PO=200
530 FOR P=0 TO 7:PRINT @(P*32)+P
O,L$(P);:PRINT CHR$(128);:NEXT P
540 GOSUB 1070
```

The next logical development is to get the figure moving, and this can be done by simply arranging to change the offset. The change must be negative as Dragons never go backwards.

```
540 PO=PO-1:GOSUB 1070:GOTO 530
```

Our Dragon will now wend his way up the screen, wrapping round at the edges, until he eventually crashes with an FC ERROR when the PRINT position goes negative. To prevent him moving upwards we could loop the offset from 31 to 0, although he will still wrap round so we now need to CLS0 at the end of each cycle, which causes the display to flash.

```
600 GOSUB 1070
610 CLS0
620 FOR PO=31 TO 0 STEP-1
630 FOR P=0 TO 7:PRINT @(P*32)+P
O,L$(P);:PRINT CHR$(128);:NEXT P
640 NEXT PO
650 CLS0
660 GOTO620
```

Horizontal screen scrolling

It is possible to produce a smooth display which scrolls without wrapping round if we add some black sections (STRING$(30,128)) either side of our Dragon pieces, and only show an incrementing MID$ section of the total string at a time.

```
700 FOR L=0 TO 7:L$(L)=STRING$(3
0,128)+L$(L)+STRING$(30,128):NEX
T L
710 CLS0
730 FOR P=1 TO 54:FOR L=0 TO 7:P
RINT @(L*32),MID$(L$(L),P,30);:P
RINT CHR$(128);:NEXT L:NEXT P
740 GOTO 730
```

Not only does the Dragon move smoothly on and off the screen, but he never reappears before he has disappeared completely. If you want to be really clever you can make sections of the screen scroll in opposite directions at the same time by making another string and modifying the way the string is sliced, although that inevitably has a time penalty.

```
720 X$=STRING$(45,128)+"FOLLOW T
HE DRAGON"+STRING$(10,128)
730 FOR P=1 TO 64:FOR L=0 TO 7:P
RINT @(L*32)+96,MID$(L$(L),P,30)
;:PRINT CHR$(128);:PRINT @0,MID$
(X$,65-P,30);:PRINT @384,MID$(X$
,65-P,30);:NEXT L:NEXT P
```

Of course this type of horizontal scrolling could be used to produce any type of design, and is particularly appropriate for large-scale displays and advertisements.

CHAPTER 8
Copying the Screen

Now that we have dealt with the commands which allow you to create graphics on the screen and looked at how we can move these around, let us consider some other features which all involve making copies of the screen.

PCOPY

The hi-resolution command PCOPY enables you to make instantaneous copies of whole graphics pages, one at a time. All you need to do is to specify the source (from) and target (to) pages. For example:

PCOPY 1 TO 2

will make a copy of graphics page 1 on graphics page 2. Remember that as this command produces a COPY the source page still has its original contents so that you can keep copying the same thing.

In PMODE0 only one graphics page is used for the whole screen display, so PCOPY can change the entire screen at one time. In the following example page 1 is first displayed and cleared to black. When a key is pressed page 2 is displayed and cleared to black. When a key is pressed page 2 is displayed and cleared to green. When a key is pressed again page 1 is copied to page 2 (the page currently being displayed) so that it changed to black.

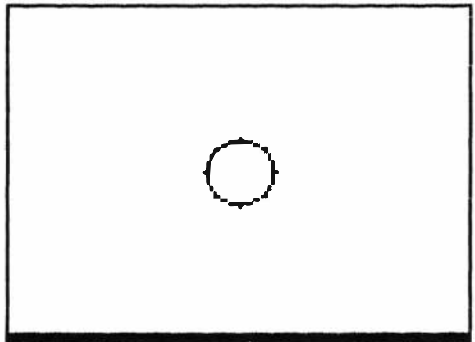
```
10 PMODE 0,1:SCREEN 1,0:PCLS
20 GOSUB 1000
30 PMODE 0,2:SCREEN 1,0:PCLS1
40 GOSUB 1000
50 PCOPY 1 TO 2
60 GOSUB 1000
999 STOP
1000 I$=INKEY$:IF I$="" THEN 100
0 ELSE RETURN
```

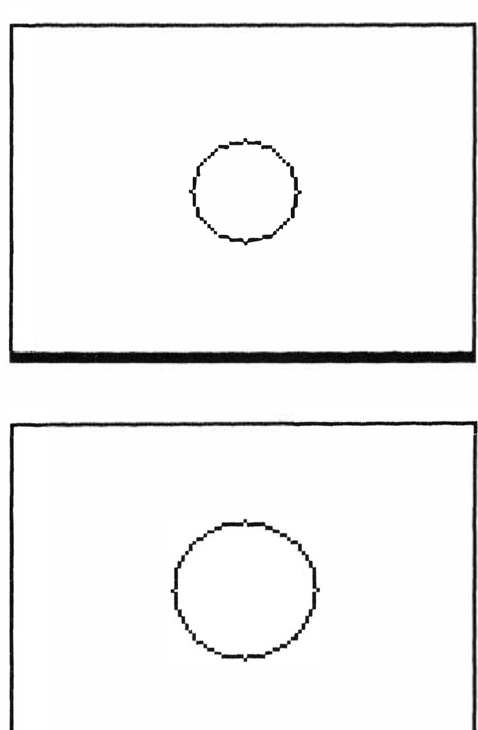
Of course you could get the same effect as this example with a simple PCLS but once you have some actual graphics on the pages the command becomes more useful, particularly in producing animation. As eight graphic pages are available, and PMODE 0 uses only one at a time, we

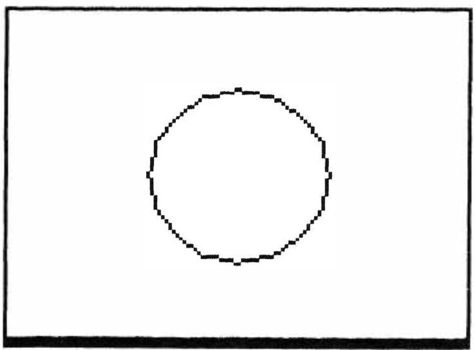
could produce seven different alternative screens containing varying sized circles on pages 2 to 8 (**Figure 8.1**) and PCOPY them back to the page on display (1) in rotation.

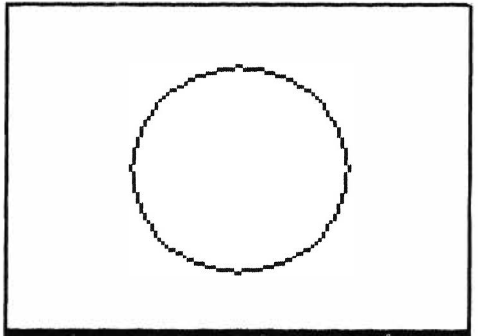
```
10 PCLEAR 8:PMODE 0,1:PCLS
20 FOR N=2 TO 8
30 PMODE 0,N:PCLS
40 CIRCLE (128,96),N*10
50 NEXT N
60 PMODE 0,1:SCREEN 1,0
70 FOR N=2 TO 8
80 PCOPY N TO 1
90 NEXT N
100 GOTO 60
```

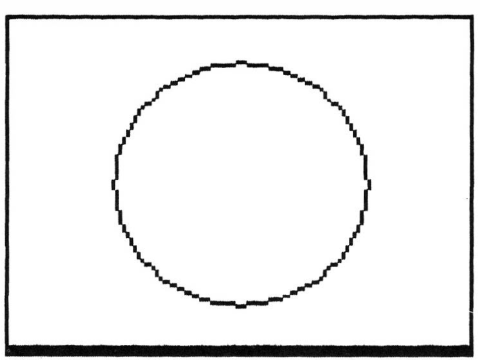
Figure 8.1

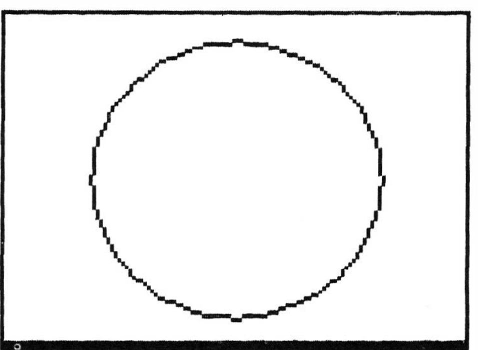


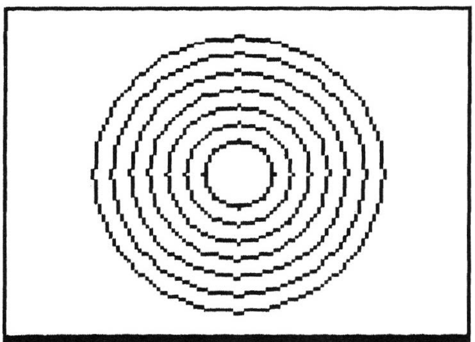












Notice that there is an initial delay while the program first draws the series of circles on pages 2 to 8. These circles are not seen until they are PCOPYed back to page 1 as the only SCREEN command follows PMODE 0,1. The very rapidly changing display almost appears to show all of the circles on screen at the same time and looks rather like a whirlpool.

(A similar effect to PCOPY can also be produced by changing the page in the PMODE command followed each time by SCREEN to change the page currently on display.

```
60 FOR N=1 TO 8
70 PMODE 0,N
80 SCREEN 1,0
90 NEXT N
```

In higher PMODEs more pages are needed for each screen so each PCOPY command will only copy part of the screen. This means that you must use more than one PCOPY command to change the whole screen, and that a smaller number of different pictures can be stored. If you modify the program to use PMODE 1 you will see that the number of alternative 'frames' is reduced from seven to three. On the other hand four colours are available, and the circles can now be drawn in three different colours.

```
10 PCLEAR 8:PMODE 1,1:PCLS
20 FOR N=3 TO 7 STEP 2
30 PMODE 1,N:PCLS
40 CIRCLE (128,96),N*10,(N/2)+1
50 NEXT N
60 PMODE 1,1:SCREEN 1,0
70 FOR N=3 TO 7 STEP 2
80 PCOPY N TO 1:PCOPY N+1 TO 2
90 NEXT N
100 GOTO 60
```

If you find it difficult to notice that the circles are different colours you can add a timing loop which will slow things down if you press a key.

```
85 I$=INKEY$:IF I$<>"" THEN FOR
T=1 TO 1000:NEXT T
```

It is not necesary to copy the whole screen at the same time and if you remove the second PCOPY in line 80 only the top half of the screen will change to produce a cycle of semicircles.

```
80 PCOPY N TO 1
```

To make this more obvious change the original PCLS on pages I and 2 to PCLS3.

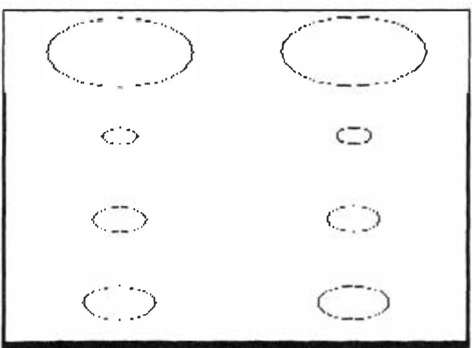
```
10 PCLEAR 8:PMODE 1,1:PCLS3
```

When you set up pictures in one mode and then view them in another, or only copy part of the screen, things can get rather confusing. If you change line 60 to PMODE 3 you may find it difficult to explain the resulting picture (**Figure 8.2**)!

```
60 PMODE 3,1:SCREEN 1,0
```

At the top you have a series of flashing coloured half-ellipses, below this a blue band, and in the bottom half of the screen a constant yellow ellipse. All of this is actually quite logical if you think about it step by step. PMODE 3 uses four pages and since we started from page 1 we can see pages 1 to 4. The blue band is page 2, which was the bottom half of the first two pages which were turned blue by PCLS3 in line 10. The yellow ellipse at the bottom was drawn on pages 3 and 4 in PMODE I, and this ellipse, and those on pages 5 to 8, are being copied back onto page 1 in the top quarter of the screen. The circles have been distorted to ellipses as each page is shortened to half of its height in PMODE I. Copying only part of the screen can be used to produce a 'window' effect where only certain sections of the screen are changed.

Figure 8.2 View in PMODE 3



In conclusion the main things to remember about PCOPY are that it is very fast but that it can only change whole pages at a time.

GETting and PUTting

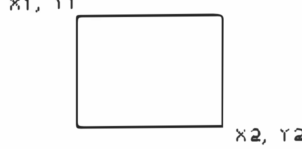
GET and PUT are an extremely valuable complementary pair of commands which allow you to store individual areas of the display and then recreate them anywhere on the screen. This is obviously a great asset when you want to move complex designs. However they are not found in most versions of BASIC and are often avoided by the novice, as they appear complex at first sight and their value and operation are not explained clearly in the Dragon manual.

GET takes an area of the screen display, and stores the status of every pixel in that area. PUT can then recreate this area anywhere on the screen. The process is rather like creating a photocopy of the GET area and PUTting it down in different places (note that the original is not affected by the copying process). The size of the area copied by GET is specified by the top-left and bottom-right coordinates of the area (**Figure 8.3**).

Array size

The pixel status information which is copied by GET must be stored in an array, and of course a suitable size array must first be DIMensioned. The method for calculating the size of the array which is described in the Dragon manual is wildly inaccurate and the figures it produces are a gross

Figure 8.3 Coordinates for GET



overestimate of those necessary. It states that you must declare a two-dimensional array using the height and the width of the area in pixel units as the dimensions. Thus for a 50 x 50 pixel area you would need a 2500 element array. Now since each array element takes up five bytes this array would occupy 12500 bytes of memory. If this was true then the GET and PUT commands would be of very limited practical use as there would only be enough room in memory for copies of quite small areas. If you calculate the array size required to GET the entire screen area of 192 x 256 pixels the result is 49,152 elements, equivalent to a horrifying quarter of a megabyte of memory!

The error in the calculation above arises from the fact that it assumes that the status of each pixel is stored as a number in a separate array element. Fortunately this is not true and information on more than one point is actually stored in each array element area. The whole idea of an array here is really rather confusing and irrelevant as in this case DIMensioning the array is only used by the system to reserve a block of memory at a particular location. The way this area is then used has no connection with the normal manipulation of arrays, as each byte is simply filled in sequence.

The amount of memory really required to store the information depends on the PMODE in operation, but let's consider the situation in the highest resolution PMODE (4) first. Here there are only two possible conditions for each screen point, as it is either on or off. Since on and off can be indicated by a single bit one byte can store the condition of eight pixels. Thus, instead of each pixel needing five bytes eight pixels can be stored in one byte — saving memory by a factor of 40! Our 50 x 50 area now only needs 313 bytes which is equivalent to a 63 element array, and even the whole screen can be stored in 1229 elements (6144 bytes). As the bytes reserved for the array are simply used in sequence there is no need to use a multi-dimension array.

The only slight problem with making the division by a factor of 40 in one step is that you can only use whole bytes and whole array elements and small errors may therefore arise due to rounding. In addition there is a

small overhead in forming the array. With small areas dividing by 40 and rounding up is always effective, but as the areas get larger some allowances must be made. The simplest way to ensure that you have enough memory reserved is to divide the area in pixel units by 40 and then add 10% to the calculation for luck. If problems arise, or you want to use the absolute minimum of memory, try RUNning the program with a slightly bigger or smaller array.

The array size needed for the other PMODEs can be calculated in the same way. Bear in mind that halving the number of individual pixels will halve the number of bits needed, but that a four colour mode needs twice the number of bits to code for pixel colour (**Table 8.1**).

Let's set up a hi-res screen and DIMension an array of appropriate size for a 50 x 50 area. The calculated array size was 63 but both the '+10%' and 'trial and error' methods show that 69 elements are needed.

```
10 PMODE4,1:SCREEN1,0:PCLS
20 DIM A(69)
1000 GOTO 1000
```

Table 8.1

CALCULATION OF ARRAY SIZE NEED BY GET

PMODE	pixels/byte	divisor (approx)
0	32	160
1	16	80
2	16	80
3	8	40
4	8	40

GET

In its simplest form GET only needs to include the top-left and bottom-right coordinates of the 'source' area and the name of the array to be filled.

GET(X1,Y1)-(X2,Y1),arrayname

```
100 GET(0,0)-(49,49),A
```

If you RUN this nothing will appear to happen, although a copy of the top left hand corner of the screen will actually have been made in array A. An optional feature in GET is the G suffix, which gives storage of full graphic detail.

GET(X1,Y1)-(X2,Y2),arrayname,graphic detail

Adding this parameter ensures that all variations of the PUT command (see below) will always work properly, but it does slow things down. The time taken to GET an area also depends on its size and **Table 8.2** gives some comparative times for different size areas, with and without the G parameter. As you might expect doubling the size of the area doubles the time taken, but the increase in time when G is added is a very significant six-fold. Much of the time G is not essential and the moral is obviously to leave it out whenever speed is important. The times for GETting the whole screen in PMODE4 are 0.6 seconds (– G) and 4.3 seconds (+ G), which can be placed in context by comparison to the speed of PCLS and PCOPY which take 0.1 seconds to change the same area.

So far we have merely been GETting a blank area of screen, so let's make a solid box in the area so that we see what is happening.

```
30 LINE(10,10)-(40,40),PSET,BF
```

Table 8.2

SPEED OF GET WITH AND WITHOUT FULL GRAPHIC DETAIL

(time in seconds)

	area (Pixels)		
	50 x 50	50 x 100	100 x 100
-G	0.04	0.08	0.14
+G	0.24	0.44	0.88

PUT

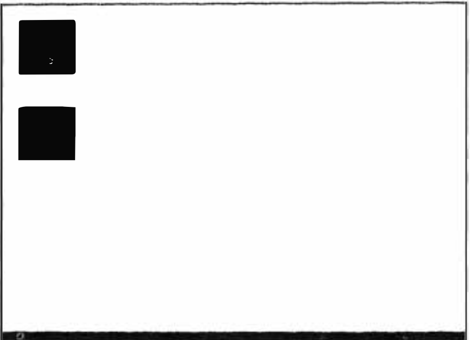
The form of PUT is similar to that for GET in that top-left and bottom-right coordinates of the 'target' area, plus the name of the array containing the information to be PUT, are the minimum requirements.

PUT(X1,Y1)-(X2,Y2),A

```
200 PUT(0,50)-(49,99),A
```

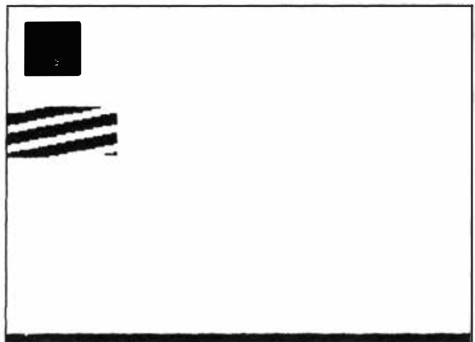
If you RUN the program so far you will see that a filled box is drawn and then this design rapidly repeated in the adjacent area (**Figure 8.4**). It is interesting to note that the original box takes 0.22 seconds to draw but the whole area is then PUT back in only 0.04 seconds. This very simple demonstration shows that it can be quicker to GET and PUT a design than to draw it from scratch and this is an important advantage of these commands.

Figure 8.4 PUT



When PUT is used as above then all points in the target area will be set to the same condition as the appropriate points in the source area, thus whatever was previously in the target area is erased completely. Care must be taken to check that the size of the area you PUT is the same as that you GET, or the copying will get out of phase. **Figure 8.5** shows the effect of an error in coordinates and a similar effect will be produced if you GET with graphic detail (G) and then PUT as described above, without specifying any of the optional actions.

Figure 8.5 PUT coordinates wrong



Actions

It is also possible to add a series of optional 'action' parameters to the PUT command.

PUT(X1,Y1)-(X2,Y2),arrayname,action

Five different actions are possible, and these only produce reliable results if you saved full graphic detail with PUT. . . . G.

The first two of these will obliterate whatever is in the target area and replace it with a copy of the source area. To see these in action modify the existing GET command to save full graphic detail, add PSET to the PUT in line 200, and add a further PUT. . . PRESET command.

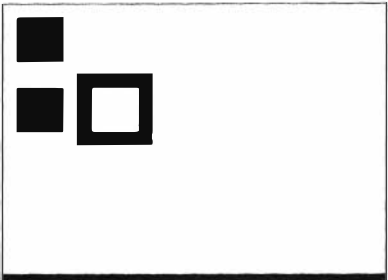
```
100 GET(0,0)-(49,49),A,G
200 PUT(0,50)-(49,99),A,PSET
210 PUT(50,50)-(99,99),A,PRESET
```

PSET has exactly the same effect as specifying no action, producing a 'positive' copy in which all points in the target area are set the same as those in the source area. The PRESET option is the inverse of PSET and produces a 'negative' of the source area by setting all points that were not set in the source array and resetting all points that were set in the source array

(Figure 8.6). As both of these wipe out the target area no superimposition occurs if you PUT back copies with overlapping coordinates **(Figure 8.7)**.

```
220 PUT( 0, 75)-(49,124),A,PSET
230 PUT(50,75)-(99,124),A,PRESET
```

Figure 8.6 PUT



PSET   PRESET

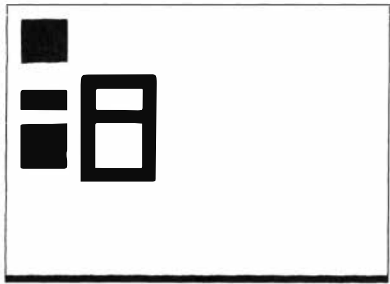
You will notice that the speed of operation of PUT has decreased markedly now that a specific action is called for. If you time it again you will find that it now takes 0.28 seconds to put a 50 x 50 area, instead of the 0.04 seconds needed when no option was chosen.

The other three options all make some form of logical comparison between the way points are set in the source and target areas. To show how they operate we will change our box to an empty version, add a series of circles to the screen, delete line 230, and then PUT the box over these circles with all the different options **(Figure 8.8)**.

```
30 LINE(10,10)-(40,40),PSET,B
40 CIRCLE(25,75),20
50 CIRCLE(75,75),20
60 CIRCLE(125,75),20
70 CIRCLE(175,75),20
80 CIRCLE(225,75),20
200 PUT(0,50)-(49,99),A,PSET
210 PUT(50,50)-(99,99),A,PRESET
```

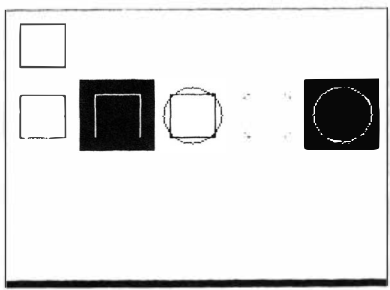
```
220 PUT(100,50)-(149,99),A,OR
240 PUT(150,50)-(199,99),A,AND
250 PUT(200,50)-(249,99),A,NOT
```

Figure 8.7 PUT with overlap



PSET PRESET

Figure 8.8 Alternative PUT actions



PSET PRESET OR AND NOT

- PSET obliterates the circle completely and produces only the box.
- PRESET gives the inverse of PSET.
- OR superimposes the source and target areas and leaves set all points which were set in either area, giving both box and circle.
- AND resets all points that were not set in both source and target areas so that only the four small areas of overlap remain set.
- NOT inverts all points in the target area, irrespective of what was in the source area. It therefore produces an inverse copy of the original contents of the target area (the circle). Notice that NOT does not fill an array, even though one must be specified to indicate the size of the area.

The value of these comparison options will become more apparent if we look at some applications.

Movement

The simplest way to produce movement is to GET without graphic detail and then PUT at coordinates which are indexed to variables, without any option. If you add this autorepeating cursor keyroutine (lines 300–340) and then RUN in PMODE 4 you will find that the up and down arrow keys work effectively along the Y axis, but that the program crashes if you try to move left or right along the X axis.

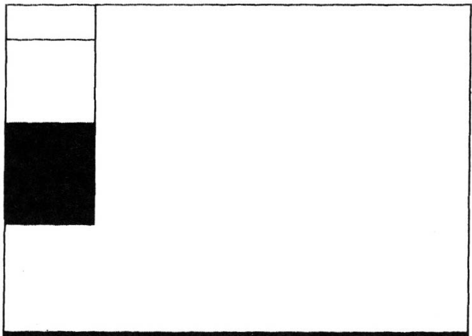
```
10 PMODE4,1:SCREEN1,0:PCLS
20 DIM A(69)
30 LINE(10,10)-(40,40),PSET,B
40 XI=1
50 YI=1
100 GET(0,0)-(49,49),A
200 PUT(X+0,Y+50)-(X+49,Y+99),A
300 IF PEEK(337)=255 THEN 300
310 IF PEEK(341)=223 THEN Y=Y-YI
:GOTO 200
320 IF PEEK(342)=223 THEN Y=Y+YI
:GOTO 200
330 IF PEEK(343)=223 THEN X=X-XI
:GOTO 200
340 IF PEEK(344)=223 THEN X=X+XI
:GOT0 200
350 GOTO 200
```

No trail is left as the step is so small that the visible area is obliterated at each move (**Figure 8.9**). If you change the coordinates of the original box to the extreme edges of the area you GET what is happening is more obvious (**Figure 8.10**).

```
30 LINE(0,0)-(49,49),PSET,B
```

Figure 8.9 No trail left if step small

Figure 8.10 Trail left if no margin



When the increment YI is 1 then a solid block follows the movement, but increasing YI (line 50) will produce less overlap until eventually if YI is greater than 50 no overlap will occur.

If you GET graphic detail and PUT with PSET the program works properly along both axes but movement is much slower (0.28 seconds/cycle instead of 0.04 seconds).

```
100 GET(0,0)-(49,49),A,G
200 PUT(X+0,Y+50)-(X+49,Y+99),A,
PSET
```

RUNning the original program which does not save all graphic detail in different PMODEs produces some strange results, although vertical movement is again no problem. The program does not crash if you move horizontally, but it does move in fits and starts in different sized steps! A little experimentation reveals that in PMODEs 4, 3 and 1 rapid but relatively smooth horizontal motion is possible provided that you are happy to change the X axis increments to steps of 8.

```
40 XI=8
```

The fastest way to update each move is thus to avoid options, although this will always leave a trail unless there is a blank space at the edges of the area you GET which is at least as large as the step size at each move. The only disadvantages of making such a blank border are that the area to be moved must be larger than the actual design and two designs cannot be placed absolutely side by side without partial erasure of one (**Figure 8.11**).

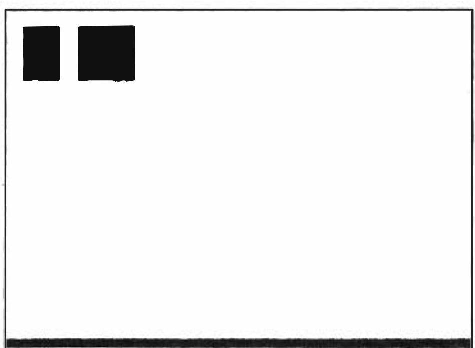
An alternative approach is to use a blank array to obliterate the old position as you move. The blank array should be the same size as the 'real' array, although, surprisingly, you do not have to use GET to fill it. The simplest method is to PUT each array in turn, although this produces a violently flashing display.

```
20 DIM A(69),B(69)
200 PUT(X+0,Y+50)-(X+49,Y+99),A
210 PUT(X+0,Y+50)-(X+49,Y+99),B
```

A more rational approach is to erase only the old area if a move is made. We could do this by moving the erasing line to a subroutine which is only called if a move is made (PEEK(337)< 255).

```
300 IF PEEK(337)=255 THEN 300 EL
SE GOSUB 500
500 PUT(X+0,Y+50)-(X+49,Y+99),B:
RETURN
```

Figure 8.11 Partial erasure if side by side



An alternative to the subroutine approach is to store the old coordinates and PUT the blank array back at these. Whatever method you use the main thing to remember is that consideration of the flow of the program is important if you are to avoid unnecessary PUT commands and thus minimise execution time.

Superimposition

The OR option is extremely valuable as it allows you to appear to move one design over (or through) another, as well as simply to superimpose them. It is not logically possible to superimpose two moving areas by using PUT. . . . OR with both arrays as there will then be no erasure of the old position in either case. The simplest method is to PUT. . . . PSET the first and then PUT. . . . OR the second, so that the first PSET clears the entire area.

The speed of execution decreases with the size of the area so that although superimposition of small areas is almost instantaneous the progress of the entire PUT sequence can be seen in larger areas.

Add the PSET option to the moving PUT of array A, draw a circle, GET this into another array (B), and then PUT it back with OR.

```
20 DIM A(69),B(69)
60 CIRCLE(75,25),20
110 GET(50,0)-(99,49),A,G
```

```
200 PUT(X+0,Y+50)-(X+49,Y+99),A,
PSET
210 PUT(50,50)-(99,99),B,OR
300 IF PEEK(337)=255 THEN 300
```

The moving box can now be placed in any position over the static circle.

Selective erasing

NOT and AND do not seem very exciting on their own but a combination of NOT and AND can be used to produce a selective erasing routine. The logical sequence is as follows:

1) GET first design into array A.
2) Invert first design with PUT. . . A,NOT
3) GET inverse version of first design into new array B.
4) Superimpose first and second designs with PUT. . . A,OR.
5) Superimpose inverse version of first design over combination of first and second designs with PUT. . . B,AND.

```
20 DIM A(69),B(69)
60 CIRCLE(125,75),20
100 GET(0,0)-(49,49),A,G
110 PUT(0,0)-(49,49),A,NOT
120 GET(0,0)-(49,49),B,G
200 PUT(100,50)-(149,99),A,OR
210 PUT(100,50)-(149,99),B,AND
```

The box appears superimposed on the circle by OR (**Figure 8.12**) and is then selectively erased completely by AND with its inverse, but only those points on the circle which are common to the first design are removed (**Figure 8.13**).

Saving Screens

In memory

It is sometimes useful to be able to store the contents of the screen so that you can recreate it later. For example you might want to be able to save a partially complete picture at some point, so that temporary alterations could be made and checked without the risk of disastrous results. The most obvious way to do this is to PCOPY the current screen page(s) onto other graphics pages. The number of complete screens that can be saved depends on the PMODE. If you set the graphics pages to the maximum (by PCLEAR 8) then in PMODEs 3 and 4 one copy can be kept, in PMODEs 1 and 2 three copies, and in PMODE 0 seven copies.

Figure 8.12 Box over circle

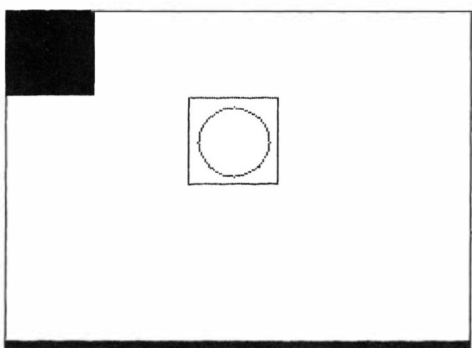
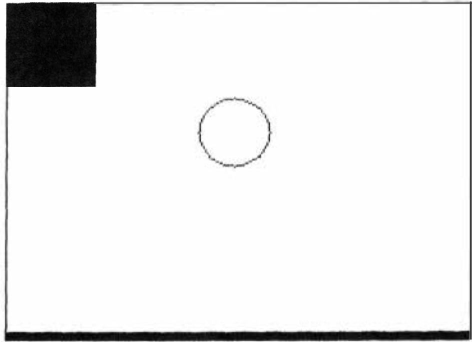


Figure 8.13 Box selectively erased



Of course you can also save only certain pages of the screen where more than one page is used. Although you cannot reserve more than eight pages

of memory for graphics it is possible to store extra copies elsewhere in memory. The easiest way to do this is simply to GET and PUT the whole screen area into an array as described earlier. This takes up the same amount of memory as the equivalent graphics page, but this memory is in the variables area and positioned above the program rather than below it. Using GET and PUT has the usual advantages that different 'actions' can be specified and any part of the screen can be saved and recreated in any position, but the disadvantage is that it is slower than PCOPY.

Permanent storage

Any screens stored in memory will be lost when the computer is turned off, but permanent copies can easily be saved on tape as a machine code file with CSAVEM. This command needs to include the start and end addresses of the area to be copied, and the number of bytes to be copied (the difference between these).

CSAVEM "name",(start),(end),(number of bytes)

These values will vary according to which PMODE is in operation and which page it starts on (**Table 8.3**).

Table 8.3

VALUES FOR CSAVEM IN DIFFERENT PMODEs

PMODE	START*	END*	NO OF BYTES
0	1536	3071	1536
1	1536	4607	3072
2	1536	4607	3072
3	1536	7679	6144
4	1536	7679	6144

* add 1536 for each start page position above page one.

For example if you have a design in PMODE 3 which starts on page 1 you can save it at any time by stopping the program with BREAK and typing this as a direct command:

CSAVEM "DESIGN", 1536,7679,6144

CSAVEM can also be included in a program so you can define variables at the start and put it into a subroutine.

```
10 N$="DESIGN":S=1536:F=7679:B=6
144

   ......GOSUB 10000

10000 CSAVEM N$,S,F,B:RETURN
```

To retrieve the design you CLOADM as a direct command or in a program line. CLOADM does not have to specify a name or any addresses and on its own it will load the next file it finds back into original position.

CLOADM (load next file onto original pages)

If you give a filename it will search and load that file.

CLOADM"DESIGN" (load file called "DESIGN" onto original pages)

The advantage of putting CLOADM in a program line is that you can set up the hi-res screen first and actually watch the screen fill.

```
10 PMODE 3,1:SCREEN 1,0
20 CLOADM
1000 GOTO 1000
```

If you want to load the file back onto different pages you can specify an "offset". As each page is 1536 bytes long each offset of 1536 will move the design up one page.

```
20 CLOADM"DESIGN",1536
```

If you RUN the above it will load back onto pages 2 to 5 instead of 1 to 4. This will proceed OK at first but your program will then vanish as you load machine code all over it! Don't forget that you must clear enough graphics pages first, by PCLEAR 5 in this case.

At first sight you may think that you should be able to save the arrays filled by GET as ASCII files on tape with PRINT # – 1. However this is not actually possible as GET does not fill the array in the normal manner but writes over all the element markers. In practice this does not really matter as you can save the screen with CSAVEM and then GET the information back into the arrays when this has been CLOADMed.

Hard copy

It is possible to print out the contents of the hi-res screen on most printers with graphics capability using only BASIC commands. In fact all the screen print illustrations in this book were produced in this way. The precise details of the program required will vary from printer to printer, as the methods of setting up graphics modes are very variable. We cannot therefore give full details, but will explain the general principles which must be applied in setting up a printer.

First the printer must be set in graphics mode (see printer manual). You cannot simply transfer the contents of each byte on the page to the printer in sequence as the bytes are mapped from left to right but the printer works on vertical segments (**Figure 8.14**). You must therefore translate the screen coordinates into vertical sections. In the byte which is transferred to the printer any bit which is 'on' will produce a printed point and any bit which is 'off' will produce a blank. Fortunately the status of each individual pixel of the display can be found by PPOINT(X,Y). The general sequence of operations is therefore to read the screen in vertical sections and set the bit if PPOINT is not zero. Setting the bit can be done by adding the appropriate number (1–128) to the byte by a logic test against PPOINT.

Figure 8.14 Comparison of screen and printer bytes

Most printers read eight vertical points at a time and this arrangement will set up the first printer byte which is taken from the top left of the screen.

```
10 X=0
100 Y=0
120 A=PPOINT(X,Y)*1+PPOINT(X,Y+1
)*2+PPOINT(X,Y+2)*4+PPOINT(X,Y+3
)*8+PPOINT(X,Y+4)*16+PPOINT(X,Y+
5)*32+PPOINT(X,Y+6)*64+PPOINT(X,
Y+7)*128
140 PRINT#-2,CHR$(A);
```

The character representation of this byte is transferred to the printer buffer by PRINT #-2,CHR$(A) but this will not be printed until a carriage return (CHR$(13)) is sent or the buffer is full. To print the first complete row we must increment X from 0 to 255, and send a CHR$(13) at the end.

```
110 FOR X=0 TO 255
150 NEXT X
160 PRINT#-2,CHR$(13);
```

To move down the screen we must increment Y in steps of 8.

```
100 FOR Y=0 TO 191 STEP 8
170 NEXT Y
```

The Seikosha GP100A is unusual in that it reads only seven bits at a time and the eighth bit must always be on so Y must be STEPed in 7's and the last bit always set.

```
100 FOR Y=0 TO 191 STEP 7
120 A=PPOINT(X,Y)*1+PPOINT(X,Y+1
)*2+PPOINT(X,Y+2)*4+PPOINT(X,Y+3
)*8+PPOINT(X,Y+4)*16+PPOINT(X,Y+
5)*32+PPOINT(X,Y+6)*64+128
```

If you want to print only part of the screen you can set appropriate limits for X and Y.

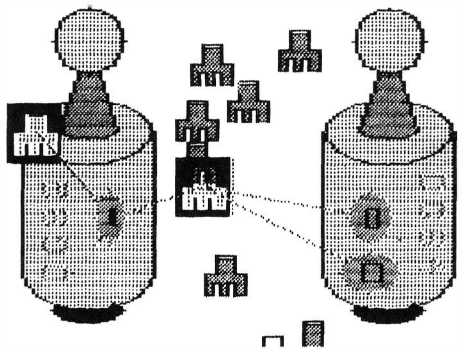
```
100 FOR Y=20 TO 100
110 FOR X=60 TO 90
```

This routine will print the screen 'as-is' — that is any bit set will be printed. This means that in a two-colour mode green or buff will be printed and black ignored. In a four colour mode the highest numbered colour will be indicated by both bits of a pair being printed, and the lowest colour by

neither being printed. The middle colours will then print either the right or left bit, producing left and right handed zebra stripes (**Figure 8.15**). It is sometimes useful to be able to invert the printing and this can be done by a change in the logic test so that ABS(PPOINT(X,Y + n) – 1) is used in place of PPOINT(X,Y + n). More complex routines can also be developed for four-colour modes which sort the colours by their PPOINT value and produce more distinctive patterns (**Figure 8.16**).

Figure 8.15 Zebra stripes when picture created in PMODE 3 printed in PMODE 4

Figure 8.16 'Four colour' print



CHAPTER 9
Graphic Presentation of Data

Bar charts

The low resolution graphics are of limited value for drawing most types of graph as they are quite crude, but they do have the advantage for bar charts that nine colours (eight plus black) and the normal text are easily available. Two colours will be needed for the background and axes, but that still leaves seven colours to indicate different things on the chart.

To demonstrate a low-res bar chart we will start by CLSl to a green background, set the first colour to 2 (yellow) and form the X and Y axes by RESETting points to black.

```
10 CLS1:C=2
30 FOR X=30 TO 60:RESET(X,25):NE
XT X
40 FOR Y=4 TO 24:RESET(30,Y):NEX
T Y
```

The X axis can be labelled by a single line.

```
50 PRINT @ 432,"1 2 3 4 5 6 7"
```

but the Y axis requires a loop which moves the print position (YP) up the screen for each value of Y (YV). YV starts at 0 and increments by (396-76)32(ie 10) for each repeat.

```
70 FOR YP=396 TO 76 STEP-32
80 PRINT @YP,YV;
90 YV=YV+10:NEXT YP
```

Now to choose seven pseudo-random values for the different bars between about 0 and 100.

```
110 DIM A(7):FOR N=0 TO 6:A(N)=R
ND(30)+(N*15):NEXT N
```

The pixels from 5 to 24 (20 positions) must represent 100 divisions hence each block will be equivalent to five units. Our array elements must be therefore converted to a number of blocks (BL).

```
170 FOR N=0 TO 6:BL=A(N)/5
200 NEXT N
```

Finally we loop up from the X axis (5) SETting the required number of blocks (BL + 5) at the appropriate Y point (29-M), moving the X coordinate (N*4 + 32) across four points and the colour up one number (C = C + 1) for each complete column.

```
180 FOR M=5 TO BL+5:SET(N*4+32,2
9-M,C)
190 NEXT M:C=C+1
210 A$=INKEY$:IF A$="" THEN 120
ELSE RUN
```

A number of things can be done to tidy up the display. First we should give the graph a title and label the axes.

```
20 PRINT @ 97,"micro-sales";:PRI
NT @ 130,"unlimited";
60 PRINT @ 469,"YEAR";
100 PRINT @227,"% PROFIT";
```

The list of numbers on the Y axis is rather ragged but it can easily be formatted with PRINT USING "###" and lined up correctly.

```
90 PRINT @YP,"";:PRINT USING"###
";YV;
```

Finally the effect is more interesting if some delay loops are included to slow down the calculations.

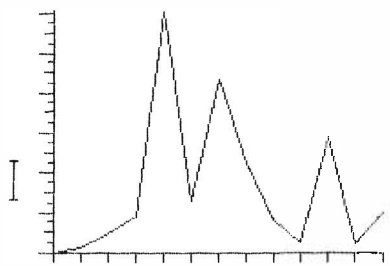
```
185 FOR T=1 TO 100:NEXT T
195 FOR T=1 TO 100:NEXT T
```

Line graphs

Line graphs can be dealt with easily in high resolution (**Figure 9.1**). First we need to set some screen coordinate limits for the start and end of the axes (XS, XE, YS, YE).

```
10 XS=40:XE=220:YS=160:YE=20
```

Figure 9.1 Line graph



The size of the X axis divisions (XI) depends on the number of readings to be fitted in. For this example we will generate a random number of readings which is in excess of five, but this would obviously normally be INPUT by the user.

```
20 NR=RND(20)+5
30 XI=(XE-XS)/NR
```

The data to be plotted needs to be entered into an array, in this case we will do it at random.

```
40 DIM A(NR)
50 FOR N=1TO NR
60 A(N)=RND(100)*RND(9)
70 NEXT N
```

A check through the array for the highest value is included and this value is then scaled down in 10% steps until it is on scale.

```
80 FOR N=1 TO NR
90 IF A(N)>HI THEN HI=A(N)
100 NEXT N
110 SF=1
120 NH=HI*SF:IF NH>(YS-YE) THEN
SF=SF-0.01:GOTO 120
```

The Y axis divisions are now set to 50 times the current scale factor (SF).

```
130 YI=50*SF
```

Now that the data is ready the hi-res screen is set up as white on black by COLOR 0,1, and the X and Y axes drawn.

```
140 PMODE4,1:SCREEN1,0:PCLS1:COL
OR0,1
150 LINE(XS,YS)-(XS,YE),PSET
160 LINE(XS,YS)-(XE,YS),PSET
```

Scale marks are set up on the X axis as a series of short LINES which are XI apart.

```
170 FOR N=XS TO XE STEP XI
180 LINE(N,YS)-(N,YS+5),PSET
190 NEXT N
```

The Y axis markings are more complex as three different length lines are used to indicate each quarter of each main division. They are placed increasing distances apart by increasing the STEP size by factors of 2. The first type of mark is most frequent and is 3 points long, the second type is 5 points long, and the last is 8 points long. The marks overwrite each other but this method is fast and is much simpler than the alternative sorting procedures.

```
200 FOR N=YS TO YE STEP -YI/2
210 LINE(XS,N)-(XS-3,N),PSET
220 NEXT N
230 FOR N=YS TO YE STEP -YI
240 LINE(XS,N)-(XS-5,N),PSET
250 NEXT N
260 FOR N=YS TO YE STEP -YI*2
270 LINE(XS,N)-(XS-8,N),PSET
280 NEXT N
```

As no text has been included in this program it is useful to have a scale line which gives a visual indication of the scale factor in use. This is formed by making a blank move to the left of the Y axis and DRAWing a scale mark there at a scale (S) which is 40 times the scale factor applied to the data.

```
290 DRAW"BM"+STR$(XS-20)+","+STR
$(YS-20)+"S"+STR$(INT(SF*40))+"B
M+0,-5L2RU10LR2S4"
```

We now need to make a blank move to the start coordinates at 0,0 assuming the graph goes through the origin (if it does not you need to make a Blank Move to the first X position and first Y array element.

```
300 DRAW"BM"+STR$(XS)+","+STR$(Y
S)
```

The graph lines are now constructed by DRAWing a MOVE to coordinates defined as the next X position and a Y value calculated from the contents of the appropriate array element multiplied by the scale factor.

```
310 FOR N=XS TO XE STEP XI
320 DRAW"M"+STR$(INT(N))+","+STR
$(INT(YS-(A((N-XS)/XI)*SF)))
330 NEXT N
340 RUN
```

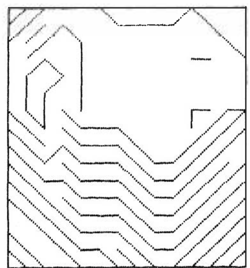
If you add the final line and RUN the program will cycle continuously through a series of randomly generated graphs.

Contour maps

A development of the line graph is the contour map (**Figure 9.2**) which links together points with the same value. This value is normally height (the example is the view through our window!) but it could just as easily be isobars on a weather chart. First we set up the screen and make a box around the map.

```
10 PMODE 4,1:SCREEN1,0:PCLS
20 LINE(10,10)-(140,160),PSET,B
```

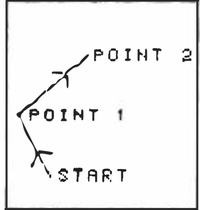
Figure 9.2 Contour map



The secret of making a simple program here is to enter the coordinate information sensibly. The DATA is taken from the map as pairs of X and Y coordinates which give the position of the next point with the same value, that is the next point to draw to (**Figure 9.3**). A pair of zeros indicates that particular line has ended and the next coordinates define the start of a new line.

```
110 DATA 1,1,1,1,0,0,1,2,2,1,0,0
,1,4,1,3,2,2,3,1,0,0,1,5,2,4,3,3
,4,2,5,1,6,1,0,0,1,6,2,5,3,4,4,3
,5,2,6,2,7,1,0,0,7,2,8,1,0,0,1,7
,2,6,3,5,4,4,5,3,6,3,7,3,8,2,9,1
,10,1,11,2,12,3,13,4,14,5,0,0,11
,1,12,2,13,3,14,4,0,0,12,1,13,2,
14,3,0,0,13,1,14,2,0,0
120 DATA 14,1,0,0,4,5,5,4,6,4,7,
4,8,3,9,2,10,2,11,3,12,4,13,5,14
,6,0,0,1,8,2,7,3,6,4,6,5,5,6,5,7
,5,8,4,9,3,10,3,11,4,12,5,13,6,1
4,7,0,0,1,9,2,8,3,7,4,8,5,7,6,7,
7,7,8,6,9,5,10,5,11,6,12,7,13,8,
14,9,0,0,1,10,2,9,3,8,0,0,4,9,5,
8,6,8,7,8,8,7,9,6,10,6,11,7
130 DATA 12,8,13,9,14,10,0,0,4,1
0,5,9,6,9,7,9,8,8,9,7,10,7,11,8,
12,9,13,10,14,10,14,11,14,12,14,
13,13,14,12,15,11,16,10,15,9,15,
8,15,7,15,6,16,5,16,4,16,3,16,2,
15,1,14,1,13,0,0,4,7,5,6,6,6,7,6
,8,5,9,4,10,4,11,5,12,6,13,7,0,0
,3,9,2,10,2,11,2,12,3,13
140 DATA 4,12,3,11,3,10,3,9,0,0,
1,11,1,12,0,0,1,15,2,16,0,0,1,16
,0,0,7,16,8,16,0,0,9,16,10,16,0,
0,12,16,0,0,14,16,0,0,14,15,0,0,
14,14,0,0,2,14,3,15,0,0,2,13,3,1
4,4,15,5,14,5,13,5,12,5,11,5,10,
0,0,11,9,11,10,12,10,0,0,11,13,1
2,13,0,0,1,16
```

Figure 9.3 Connecting points on contour map



The start position is first READ, a Blank Move made to this, and the pointer RESTOREd to the start of the DATA.

```
30 READ X,Y:DRAW"BM"+STR$(X*10)+
","+STR$(170-(Y*10)):RESTORE
```

The 175 pairs of DATA points are now read in turn. If X and Y are not zero then a MOVE is made (line drawn) to coordinates calculated by X and Y multiplied by 10, and then the next points READ. As the first DATA was RESTOREd the first line is of zero length to and from the first point. If a zero is READ then the next values of X and Y are READ, a Blank Move made to those coordinates, and the next values READ.

```
40 FOR N=1 TO 175
50 READ X,Y
60 IF X=0 THEN READ X,Y:DRAW"BM"
+STR$(X*10)+","+STR$(170-(Y*10))
:NEXT N:GOTO 90
70 DRAW"M"+STR$(X*10)+","+STR$(1
70-(Y*10))
80 NEXT N
90 GOTO 90
```

Pie charts

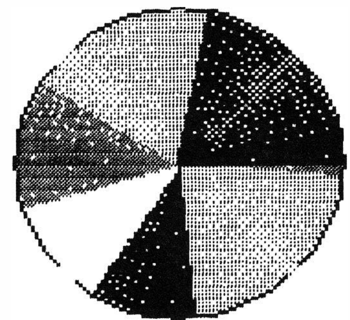
Circular pie charts in which the size of the slices is the indication of quantity can be easily produced in high resolution. First we need to call a suitable four-colour hi-res mode, set up an array to hold our values, and set the first colour to 1. Pie charts are usually divided into quite a small number of slices so we will take a series of seven random numbers. Note that the total (T) also needs to be calculated.

```
10 PMODE 1,1:SCREEN1,0:PCLS
20 DIMSL(7):C=1
30 FOR N=1 TO 7:SL(N)=RND(10):T=
T+SL(N):NEXT N
```

Now we can draw a circular outline, and then a series of arcs of increasing radius and differing length to indicate the slices **(Figure 9.4)**. As explained earlier the arcs will not be completely filled, but they are still quite effective, and very simple to construct. The filling is more complete in PMODE 1 than in PMODE 3.

```
40 CIRCLE(128,96),90
50 FOR S=1 TO 7
60 FI=ST+(SL(S)/T)
70 FOR R=1 TO 88
80 CIRCLE(128,96),R,C,1,ST,FI
90 NEXT R
100 ST=FI:C=C+1:IF C>4 THEN C=1
110 NEXT S
120 A$=INKEY$:IF A$="" THEN 120
ELSE RUN
```

Figure 9.4 Seven slice pie chart starting from 0 (90°)



The default start (ST) value will be 0 so that the first arc will move clockwise from 3 o'clock. The finish (FI) of the arc is calculated in appropriate units by dividing the slice value in the array (SL(S)) by the total (T) and adding this to the start value. After each arc is plotted the new start value (ST) is set to the old finish point (FI), and the colour is incremented by one. If the colour is greater than 4 it is reset to 1. The largest ARC is slightly smaller (R = 88) than the radius of the CIRCLE (R = 90) so that the slice formed in the background colour does not erase the outline.

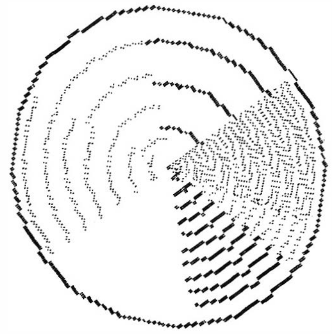
If you want the slices to start from 12 o'clock you need to set ST to 0.75 at the start (if values larger than 1 are generated the integer part is ignored).

```
20 DIMSL(7):C=1:ST=0.75
```

As only four colours are available it is at first sight difficult to indicate more than four different slices but one way to ensure that even slices with the same colour are distinctive is to change the step size of the arcs if more than four slices are to be plotted and it is also useful to link the step size to the colour number (**Figure 9.5**).

```
20 DIMSL(7):C=1:SP=2:ST=0.75
60 FI=ST+(SL(S)/T):IF S>4 THEN S
P=6
70 FOR R=1 TO 88 STEP SP*C
```

Figure 9.5 Seven slice pie chart starting from 0.75 with step size linked to slice number

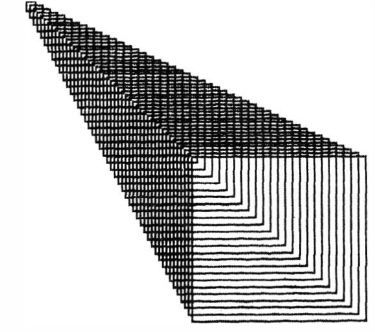


CHAPTER 10
Three Dimensions

Presenting a three-dimensional view of an object is a very effective way of making it look more solid. The important thing about three-dimensional representation is that lines which are supposed to be further away are drawn smaller. For example a looping program which draws a series of boxes which get larger and are offset slightly each time gives the impression of a square cone (**Figure 10.1**).

```
10 PMODE 4,1:SCREEN 1,0:PCLS
20 Y0=0:X0=5
30 FOR X=10 TO 100 STEP 2
40 LINE(X,Y)-(X+R,Y+R),PSET,B
50 Y0=Y0+2:X0=X0+2
60 NEXT X
520 A$=INKEY$:IF A$="" THEN 520
530 RUN
```

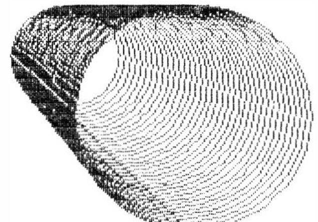
Figure 10.1 3-D Box Section



An even more real tube effect is produced by drawing a series of offset CIRCLES of increasing radius (**Figure 10.2**).

```
20 YO=50:R=30
30 FOR X=50 TO 120 STEP 2
40 CIRCLE(X,YO),R
50 YO=YO+1:R=R+1
```

Figure 10.2 Tube



If we set up some limits for X and Y axes (XS = X start, XE = X end, YS = Y start, YE = Y end) we can draw a rectangle by a series of lines connecting these points (**Figure 10.3**).

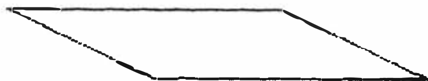
```
40 XS=100:XE=250
60 YS=160:YE=120
210 PMODE4,1:SCREEN1,0:PCLS
230 LINE(XS,YS)-(XE,YS),PSET
240 LINE(XS,YE)-(XE,YE),PSET
260 LINE(XS,YS)-(XS,YE),PSET
270 LINE(XE,YS)-(XE,YE),PSET
```

Figure 10.3 Lines connected in rectangle

To make this appear as a flat surface in three dimensions we must displace the back edge by some distance to one side (DI) (**Figure 10.4**).

```
80 DI=80
240 LINE(XS-DI,YE)-(XE-DI,YE),PS
ET
260 LINE(XS,YS)-(XS-DI,YE),PSET
270 LINE(XE,YS)-(XE-DI,YE),PSET
```

Figure 10.4 Displacement to indicate depth



Although this looks 'flatter' it is still not quite correct as the back edge is the same length as the front edge. It is the line from (XS–DI,YE) to (XE–DI,YE) (line 240) which needs shortening at one end and a little experimentation will reveal a value for this perspective factor (PF) which 'looks' right. You must not forget that line 270 must also be modified.

```
80 DI=80:PF=25
240 LINE(XS-DI,YE)-(XE-DI-PF,YE)
,PSET
270 LINE(XE,YS)-(XE-DI-PF,YE),PS
ET
```

If you want to produce a three-dimensional graph you can usually actually get away with forgetting about getting the perspective exactly correct, especially if you leave out the top horizontal and right vertical lines, so delete the perspective factor and lines 240 and 270 (**Figure 10.5**). We need some data to plot so let's generate some at random.

```
100 DIM A(15,15)
110 FOR N=0 TO 15
120 FOR M=0 TO 15
130 A(N,M)=INT(RND(N*3)+20+RND(M
*3))
140 NEXT M
150 NEXT N
```

Figure 10.5 X and Y axes

That produces 225 numbers in a 15 × 15 two-dimensional array which we will use to indicate height. To produce a 3-D graph linking these points we need to define the divisions on the X (XI) and Y (YI) axes, and arrange to step through the coordinates. The X loop is outside the Y loop so we will move first from front to back.

```
20 XI=20:YI=10
290 FOR N=XS TO XE STEP XI
330 FOR M=YS TO YE STEP-YI
```

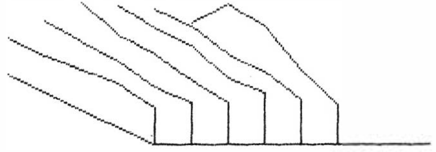
The step is negative for YI as we want to work from front to back. The correct X axis array element (XP) is selected by dividing the current N minus the start position by the size of the divisions.

```
300 XP=(N-XS)/XI
```

We must next make a blank move to the first position, calculate the next Y array element and move (draw a line) to the point defined in that element (**Figure 10.6**).

```
310 DRAW"BM"+STR$(N)+","+STR$(YS
)
320 D=0
340 YP=(YS-M)/YI
350 DRAW"M"+STR$(N-D)+","+STR$(M
-A(XP,YP))
```

Figure 10.6 Front to back moves



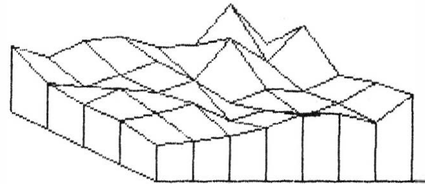
The X axis element must take into account how much displacement (D) to the left must be made. For the first point this is set to 0, but for subsequent points it must be calculated from YI.

```
370 D=D+(YI*2)
380 NEXT M
390 NEXT N
```

At the end of the first front-back line the X position is incremented and the next line drawn. The left-right lines are drawn in a similar way (**Figure 10.7**).

```
410 D=0
420 FOR M=YS TO YE STEP-YI
430 YP=(YS-M)/YI
440 DRAW"BM"+STR$(XS-D)+","+STR$
(M)
450 FOR N=XS TO XE STEP XI
460 XP=(N-XS)/XI
470 DRAW"M"+STR$(N-D)+","+STR$(M
-A(XP<YP))
490 NEXT N
500 D=D+(YI*2)
510 NEXT M
```

Figure 10.7 Plus left to right

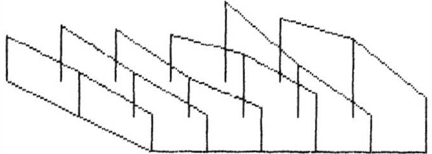


If you want to include vertical lines to indicate height you can add these lines which DRAW with N (no update) so that the last position is remembered (**Figure 10.8**).

```
360 DRAW"NM"+STR$(N-D)+","+STR$(
M)
```

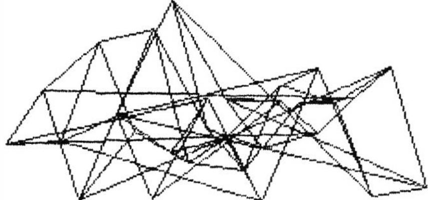
```
480 DRAW"NM"+STR$(N-D)+","+STR$(
M)
```

Figure 10.8 Vertical lines included



It is also possible to plot in three dimensions without ever showing the axes (**Figure 10.9**).

Figure 10.9 Plot with axes omitted

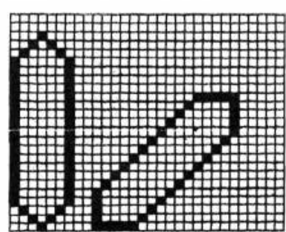


CHAPTER 11
Rotation of Figures

Using angled draw command

The simplest type of rotation is catered for directly in the BASIC DRAW command in which the angle can be specified from 0 to 3, to give four copies transformed at 90 degrees. In effect this means that at each turn U is read as R, R as D, D as L and L as U, etc. At first sight you may think that you could replace these directly with the diagonal DRAW parameters (E, F, G and H) to give the intermediate positions, but life is not that simple (**Figure 11.1**) shows two versions of the same design which differ by an angle of 45 degrees and if you look closely you will see that the number of pixels needed to make each section of the same design is actually different in the two cases. This is because all these commands move an absolute number of pixel units, but the hypotenuse of an isosceles triangle is actually almost 1.5 times as long as the other two sides. Thus three units Up, Down, Left or Right mean the same actual distance on the screen as two units of E, F, G or H. In **Figures 11.2** and **11.3** a circle with a radius the length of the design has been made around alternative figures made with equal numbers of pixel units and with the number of units corrected for the difference in direction.

Figure 11.1 'Draw'ing at an angle



This program rotates the two alternate pictures through all the possible positions, and gives the effect of motion by drawing in foreground and then background colour.

```
10 PMODE 4,1:SCREEN 1,0:PCLS
20 A$="H3U16E3F3D16G3"
30 B$="U4E11R4D4G11L4"
40 FOR A=0 TO 3
50 FOR C=1 TO 0 STEP-1
60 DRAW "A"+STR$(A)+"C"+STR$(C)+
A$
70 NEXT C
80 FOR C=1 TO 0 STEP-1
90 DRAW "C"+STR$(C)+B$
100 NEXT C
110 NEXT A
120 GOTO 120
```

Figure 11.2 Equal numbers

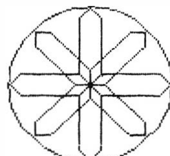
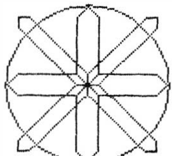


Figure 11.3 Corrected



Using mathematics

To mathematicians all things are possible (they tell us), even if we cannot understand why and how. If you are really interested in producing complex rotations then you are going to have to brush up your knowledge of trigo-

nometry and matrices and also find a good book on the subject. However, as an introduction we will look at how to rotate a figure in two dimensions, as this is quite easy. First you need to set up the screen and define the point about which you want to rotate (XS,YS). A small cross is formed to mark this position.

```
30 PMODE 4,1:SCREEN 1,0:PCLS
40 XS=128:YS=96
90 DRAW"BM"+STR$(XS)+","+STR$(YS
)+"U4D2L2R4"
```

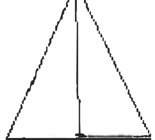
Now we need something to rotate, so let's form a bisected triangle pointing upwards. This is made by connecting four points and we define each of these in terms of the number of screen points they are away from the centre of rotation along the X (P1(N)) and Y (P2(N)) axes, anything to the left or up being negative. We will put these into an array as it makes the program for the calculation of each point neater, and then PSET them.

```
20 DIM P1(4),P2(4)
50 P1(1)=0:P2(1)=0
60 P1(2)=0:P2(2)= -60
70 P1(3)= -30:P2(3)=0
80 P1(4)=30:P2(4)=0
120 FOR N=1 TO 4
150 PSET(XS+P1(N),YS+P2(N))
160 NEXT N
230 GOTO 230
```

To connect the points together we can form LINEs, and as we must connect them in the order of 1 to 2, 2 to 3, 3 to 4 and 4 to 2 (**Figure 11.4**) we will put this order in a DATA statement and READ it back from there.

```
10 DATA 1,2,2,3,3,4,4,2
170 FOR L=1 TO 4
180 READ N1,N2
190 LINE(XS+P1(N1),YS+P2(N1))-(X
S+P1(N2),YS+P2(N2)),PSET
200 NEXT L
210 RESTORE
```

Figure 11.4 Lines connected to form bisected triangle



Now for what looks like the hard bit, calculating the new positions for each point when it is rotated through a certain angle. The rules are:

1) The angle must be given in radians, so degrees must first be converted.

```
100 FOR AN=1 TO 360 STEP 90
110 A=AN*3.142/180
220 NEXT AN
```

2) The new X and Y axis positions (NX(N) and NY(N)) for each old point are calculated by the following two formulae.

```
130 NX(N)=P1(N)*COS(A)+P2(N)*SIN
(A)
140 NY(N)= -P1(N)*SIN(A)+P2(N)*C
OS(A)
```

Note the minus sign in the second line and also remember that these are still only displacements from the centre of rotation so we must add this back on to find the actual screen positions.

The STEP in angle included above is 90 degrees so the triangle will be constructed in four alternative positions (**Figure 11.5**). If you reduce the STEP the number of positions increases and the result can become very complex (**Figure 11.6**) and yet another way of generating patterns.

Figure 11.5 Triangle rotated through 360° in 90° steps

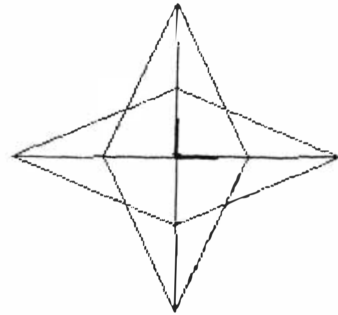
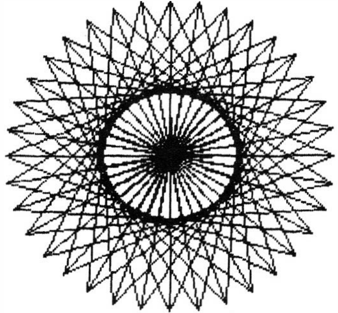
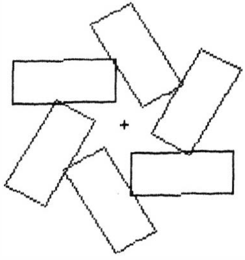


Figure 11.6 Triangle rotated through 360° in 10° steps



The design you rotate can be of any shape, and does not have to touch the centre of rotation. Work out for yourselves what modifications you need to make to the original coordinates and DATA to produce the picture in **Figure 11.7**.

Figure 11.7 Rectangle rotated away from centre



CHAPTER 12

Instant Keyboard Access to Hi-Res Commands

Although we have already explained in detail how to use each of the hi-res graphics commands in your programs, all your efforts have had to be planned in advance. On the other hand drawing directly on the screen can be very useful as you can change your ideas easily as you go. If you try to use INPUT in hi-res you revert to the text screen as INPUT halts the program. On the other hand INKEY$ can be used, the simplest form of 'direct drawing' scanning INKEY$ for letters which can be used in a DRAW string.

```
10 PMODE 4,1:SCREEN 1,0:PCLS
20 A$=INKEY$:IF A$="" THEN 20
30 DRAW A$
40 GOTO 20
```

RUN this and you will find that each time you press a valid key (U, D, L, R, E, F, G or H) you will DRAW Up, Down, Left, Right or in one of the four diagonal directions.

Of course that is of limited practical value as you can only DRAW in simple ways, and cannot even MOVE without drawing, so now we will consider how to build up a sophisticated direct drawing program which allows you to manipulate all the graphics commands directly from the keyboard whilst viewing the hi-res screen. This relies on keychecks using INKEY$ and also certain PEEKs to the keyboard scan routines, provides single key definition of graphics commands, and extensive use of GET and PUT.

Setting up

The first task is to set up the screen with the required PMODE, SCREEN, and foreground and background COLORs.

```
10 CLS:PRINT"PMODE":INPUT P:PRIN
T"COLOUR SET":INPUT SN:PRINT"FOR
EGROUND COLOUR":INPUT C1:PRINT"B
ACKGROUND COLOUR":INPUT C2:PRINT
```

```
"X START AND END":INPUT XS,XE:PR
INT"Y START AND END":INPUT YS,YE
:GOTO 5000
5000 PMODE P,1:SCREEN 1,SN:PCLS
C2:COLOR C1,C2:DRAW"C"+STR$(C1)
```

A series of arrays must be DIMensioned to hold various areas to be taken by GET commands. These are each described in detail later. The start position is defined by X and Y as screen centre (128,96), and the cursor increment (IN) set to 4. X and Y are kept updated in the program and always indicate the current screen position. Finally a list of the single keys which will be used to access the graphics commands is made in VK$. Note that there is a space between W and S, and two keys are defined by their CHR$ codes. These are CLEAR and ENTER, which are not displayable characters.

```
5010 DIM CU(10):DIM OC(10):DIM S
C(1229):DIM WM(200):DIM GP(1229)
:DIM PR(10):X=128:Y=96:IN=4:VK$=
"RBCEFGLNPQJ1234KOW SODT"+CHR$(1
2)+CHR$(13)+"IA"
```

We can now go back to the main keycheck routine at 1000.

```
5020 GOTO 1000
```

Keycheck routine

Repeated movement in the same direction is easier if keys autorepeat. To give autorepeat the main keycheck routine looks at locations 337 and 135 instead of using INKEY$. The keys are sorted by comparing the CHR$ code of PEEK(135) against the list made in VK$ using INSTR. The value of K will depend on the position of the character in the list, and will lead to an appropriate subroutine.

```
1000 IF PEEK(337)=255 THEN 1130
1010 A=PEEK(135)
1020 A$=CHR$(A)
1030 K=INSTR(1,VK$,A$)
1040 ON K GOTO 1200,1300,1400,15
00,1600,1700,1800,1900,2000,2100
,2200,2300,2400,2500,2600,2700,2
800,3000,3200,3300,3400,3500,360
0,3700,3800,3900,4000
```

If the key pressed is not in this list control falls through to the cursor key check. A logical test for both shifted and unshifted keys is made, and the X and Y coordinates updated by the current size of increment (IN) as appropriate. Checks are included to ensure that the limits of the defined screen area are not exceeded.

```
1050 Y=Y+(IN*((A=94)-(A=10)))
1060 Y=Y+(IN*((A=95)-(A=91)))
1070 IF Y>YE THEN Y=YE
1080 IF Y<YS THEN Y=YS
1090 X=X+(IN*((A=8)-(A=9)))
1100 X=X+(IN*((A=21)-(A=93)))
1110 IF X>XE THEN X=XE
1120 IF X<XS THEN X=XS
```

Cursor

The cursor needs to be totally non-destructive or it will erase part of the design on the screen. A very small non-destructive cursor can be produced by reading a pixel with PPOINT and then PSETting, as described earlier, but a better way is to GET and PUT an area of the screen around the current position. (This conservative (with a small c) GET and PUT technique is also used extensively elsewhere in this program.) Any size cursor can be produced but 2 pixels by 2 pixels is convenient. We GET this into the cursor array (CU), with graphic detail, and this array is immediately PUT back with PRESET which inverts the display at that point. After a short time delay the original screen is recreated by PUTting back array CU with PSET. The overall effect of this is a rapidly flashing square cursor.

```
1130 GET(X-1,Y-1)-(X+1,Y+1),CU,G
1140 PUT(X-1,Y-1)-(X+1,Y+1),CU,P
RESET
1150 FOR N=1 TO 10:NEXT
1160 PUT(X-1,Y-1)-(X+1,Y+1),CU,P
SET
```

One is subtracted from the start limits for X and Y to prevent the cursor trying to reach illegal negative coordinates when moved to the extreme top or left.

Moving and drawing

If you RUN this you will now find that the unshifted arrow keys will move the cursor around the screen, but that no trail is left.

However we really need to produce two different possibilities, moving without drawing, and moving with drawing. If we test location 337 against

both 159 AND 191 we can distinguish that one of the shifted cursor keys has been used, whilst still retaining the autorepeat. If an unshifted key has been pressed then a Blank Move is made to the new X and Y coordinates, but if a shifted key is used a MOVE is made, thus drawing a line to the new X and Y coordinates. The line is drawn in the current foreground colour.

```
1170 IF PEEK(337)<>159 AND PEEK(
337)<>191 THEN DRAW"BM"+STR$(X)+
","+STR$(Y):GOTO 1000
1180 DRAW"M"+STR$(X)+","+STR$(Y)
:GOTO 1000
```

RUN again and note the difference between the shifted and unshifted keys.

Single key routines

A whole series of graphics routines can be called by pressing a single key. Wherever possible the key is used as a mnemonic(prompt) for the action. The routines vary widely in their complexity, so let's start with something very simple.

Cursor increment

The distance moved by the cursor in each cycle is controlled by the increment IN which is originally set to 4. The keys 1 to 4 are designated to give four alternative values for IN of 1, 2, 4 and 8.

```
2300 IN=1:GOTO 1000
2400 IN=2:GOTO 1000
2500 IN=4:GOTO 1000
2600 IN=8:GOTO 1000
```

Try RUNning again and see the effect of pressing keys 1 to 4 on the rate of movement and drawing.

Leaving your mark

Whilst shifted arrows can be used to DRAW lines this can become tedious, especially for long distances. It would therefore be convenient to be able to use commands such as LINE, but of course these require the coordinates of both ends of the LINE to be specified. To use these we must leave a marker at the start of the LINE and then move the cursor to the end point. First we

indicate that we want to drop a start marker by pressing the space bar to reach the subroutine at 3200.

```
3200 IF CF=1 THEN PUT(XO-1,YO-1)
-(XO+1,YO+1),OC,PSET:GOTO 1000
3210 XO=X:YO=Y:CF=1
3220 GET(XO-1,YO-1)-(XO+1,YO+1),
OC,G
3230 PUT(XO-1,YO-1)-(XO+1,YO+1),
OC,PRESET
3240 GOTO 1000
```

The first line is skipped the first time through as CF has the default value of 0, so the 'old' X and Y positions are read into XO and YO and the cursor flag (CF) set to 1. The screen contents at the old position are now taken with GET into the old cursor (OC) array, and PUT back PRESET to leave an inverse square at the origin. You return to the keycheck routine and can move the flashing cursor as before until you reach a point where you want to make another decision.

If you change your mind and decide that you want to erase your marker without using it just press the spacebar again. As CF is now 1 you will erase your mark.

Line

Pressing L indicates that you want to draw a LINE from the mark (XO,YO) to the current cursor position (X,Y). If you have not made a mark CF will be 0 and you jump straight back. You must also PUT back the screen display at the old cursor position and reset CF.

```
1800 IF CF=0 THEN 1000
1810 PUT(XO-1,YO-1)-(XO+1,YO+1),
OC,PSET
1820 LINE(XO,YO)-(X,Y),PSET
1830 CF=0:GOTO1000
```

Rubout

If you decide that your Line was a mistake you can use R to rub it out by means of LINE with PRESET in the same way.

```
1200 IF CF=0 THEN 1000
1210 PUT(XO-1,YO-1)-(XO+1,YO+1),
OC,PSET
1220 LINE(XO,YO)-(X,Y),PRESET
1230 CF=0:GOTO 1000
```

No-update

It is sometimes convenient to form a series of LINEs which radiate from a central point. The routine for this is reached through N and is even simpler as you just draw from the old cursor position to the new but do not erase the old mark.

```
1900 IF CF=0 THEN 1000
1910 LINE(X0,Y0)-(X,Y),PSET
1920 GOTO 1000
```

If you drop a mark, move to a series of different positions, and press N at each point you will produce a series of lines.

Finally press the spacebar to erase the old position marker or use 'L'ine for the last line.

Box and filled box

As both empty and filled boxes are formed by adding suffixes to the LINE command these can be produced by similar routines. A mark is dropped and then the cursor moved to the diagonally opposite corner and B or F pressed.

```
1300 IF CF=0 THEN 1000
1310 PUT(X0-1,Y0-1)-(X0+1,Y0+1),
OC,PSET
1320 LINE(X0,Y0)-(X,Y),PSET,B
1330 CF=0:GOTO 1000
```

```
1600 IF CF=0 THEN 1000
1610 PUT(X0-1,Y0-1)-(X0+1,Y0+1),
OC,PSET
1620 LINE(X0,Y0)-(X,Y),PSET,BF
1630 CF=0:GOTO 1000
```

Circle

To produce a CIRCLE we need to define the centre and the radius. The centre is marked as before, the cursor moved out to the edge of the proposed CIRCLE, and C pressed.

```
1400 IF CF=0 THEN 1000
1410 PUT(X0-1,Y0-1)-(X0+1,Y0+1),
OC,PSET
1420 R=SQR((ABS(X0-X)^2)+(ABS(Y0
-Y)^2))
```

```
1430 CIRCLE(XO,YO),R
1440 CF=0:GOTO 1000
```

It does not matter in which direction you mark the radius (R) as it is calculated as the hypotenuse of a right-angled triangle formed by the ABSolute differences in X and Y coordinates between the mark and cursor positions. The circle will be drawn in the current foreground colour, but this can also be changed by a single key command to give different coloured CIRCLEs.

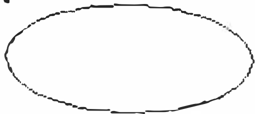
Ellipses

Ellipses are just varieties of CIRCLEs as far as the computer is concerned, but you must now specify both Width and Height separately so some modification is necessary.

```
1500 IF CF=0 THEN 1000
1510 PUT(XO-1,YO-1)-(XO+1,YO+1),
OC,PSET
1520 W=ABS(XO-X):H=ABS(YO-Y)
1530 CIRCLE(XO,YO),W,,H/W
1540 CF=0:GOTO 1000
```

To form an ellipse mark the centre, move to a point which is half the width of the ellipse away along the X axis, and half the height away along the Y axis (**Figure 12.1**), and then press E.

Figure 12.1 Cursor in position to mark height and width of ellipse



GET

Any area of the screen can be stored in an array by reaching the GET routine via G. The size of the array originally set up will take the entire screen so there should be no problems. The screen box to be taken is

marked as for a normal box, except that you must mark the top-left and bottom-right positions in that order.

```
1700 IF CF=0 THEN 1000
1710 PUT(XO-1,YO-1)-(XO+1,YO+1),
OC,PSET
1720 GET(XO,YO)-(X,Y),GP,G
1730 GX=XO-X:GY=YO-Y:CF=0
1740 SOUND 255,1:GOTO 1000
```

The size of the box along the X and Y coordinates must be recorded as GX and GY so that it can be PUT back correctly.

GET is very useful for producing copies of a single picture elsewhere on the screen, or for experimenting with different positions for a design.

Backup

As your designs become more complex you become increasingly afraid that you will make a disastrous irrevocable error. To guard against this you can include a 'backup' facility which you can use at any point by pressing 'CLEAR'. This facility also gives you the ultimate in 'rubber-banding' (or what happens if?) as you can store the screen and actually find out.

```
3700 GET(XS,YS)-(XE,YE),SC
3710 SOUND 255,1:GOTO 1000
```

This GETs a copy of the entire work area of the screen into array SC, gives a signal, and returns. To PUT this screen back at any point press ENTER.

```
3800 PUT(XS,YS)-(XE,YE),SC
3810 SOUND 255,1:GOTO 1000
```

Of course this routine will only store one screen as each time you use this copying process you overwrite the old screen you stored, but it is invaluable for temporary storage if your nerves are bad. Get into the habit of pressing CLEAR when you can't think what to do next, and before disaster strikes. If you really need two backup copies of the whole screen you could GET another one into the GP array.

PUT

All the different options of PUT can be used to recreate the area taken by GET into the GP array at any screen position with any action. The top left coordinates are the cursor position and the others are calculated from the record of the size of the area (GX and GY).

P gives PSET

```
2000 PUT(X,Y)-(X-GX,Y-GY),GP,PSE
T
2010 SOUND 255,1:GOTO 1000
```

I gives PRESET (Inverse)

```
3900 PUT(X,Y)-(X-GX,Y-GY),GP,PRE
SET
3910 SOUND 255,1:GOTO 1000
```

D gives AND

```
3500 PUT(X,Y)-(X-GX,Y-GY),GP,AND
3510 SOUND 255,1:GOTO 1000
```

O gives OR

```
3400 PUT(X,Y)-(X-GX,Y-GY),GP,OR
3410 SOUND 255,1:GOTO 1000
```

T gives NOT

```
3600 PUT(X,Y)-(X-GX,Y-GY),GP,NOT
3610 SOUND 255,1:GOTO 1000
```

If the coordinates fall off the screen the recreation will be corrupted, so it is best to use the backup routine first if you are close to the bottom or right hand side.

Kill

If you want to abandon the current effort and PCLS press K. As this is a permanent action which must not be called by accident certain safeguards are built in. A non-destructive inversion of the top of the screen is given as a warning, and K must be held down for 5 cycles within 6 seconds for a PCLS to be carried out.

```
2700 GET(0,0)-(255,10),WM,G
2710 PUT(0,0)-(255,10),WM,PRESET
2720 IF CL=0 THEN TIMER=0:CL=CL+
1: ELSE CL=CL+1:IF TIMER>300 THE
N CL=0:ELSE IF CL=5 THEN PCLS C2
```

```
:CL=0
2730 PUT(0,0)-(255,10),WM,PSET
2740 GOTO 1000
```

When you press K you GET a band at the top of the screen into array WM (warning mark), and PUT it back PRESET (inverted). The first time through the TIMER is reset, the clear flag (CL) incremented by 1, and WM PUT back PSET. If K is still pressed the TIMER is checked against 300, and if this value is reached the clear flag is reset. If CL has counted up to 5 then a PCLS occurs.

Change colours

Nowthat we have started running out of suitable keys we will haveto use W to indicate Which colours to use for foreground and background. As we have no text available on the screen so far indications of what stage you are at is given by moving inverted blocks across the top of the screen. The first block is inverted towards the left of the screen indicating that the foreground colour is to be entered as C1$. As we need to use the non-repeating INKEY$ here the autorepeat must be disabled by POKE 135,0. Once the foreground colour has been entered the original block is PUT back, an inversion made on the right of the screen, and the background colour entered as C2$. Finally this block is PUT back and the COLORs changed by taking the VAL of C1$ and C2$ and DRAWing "C" +C1$.

```
3000 GET(40,0)-(50,10),PR,G
3010 PUT(40,0)-(50,10),PR,PRESET
3020 FOR N=1 TO 1000:NEXT N
3030 POKE 135,0
3040 C1$=INKEY$:IF C1$="" THEN 3
040
3050 PUT(40,0)-(50,10),PR,PSET
3060 GET(168,0)-(178,10),PR,G
3070 PUT(168,0)-(178,10),PR,PRES
ET
3080 C2$=INKEY$:IF C2$="" THEN 3
080
3090 PUT(168,0)-(178,10),PR,PSET
3100 C1=VAL(C1$):C2=VAL(C2$):COL
OR C1,C2:DRAW"C"+C1$:GOTO 1000
```

Paint

Paint coordinates and colours are entered in a similar way, but as PAINTing often causes unexpected results a backup copy is first automatically made by calling the routine at 3700, and the Paint Flag is also set to 1. The

cursor is set on the point to start PAINTing from, and Q pressed. A block is displayed at the left, and the first colour entered. The second block indicates the second (border) colour, and the final block asks for confirmation of your decision. If PA$ is not Y then the PAINTing is abandoned.

```
2800 IF PF=0 THEN PF=1:GOTO 3700
2810 GET(0,0)-(10,10),PR,G
2820 PUT(0,0)-(10,10),PR,PRESET
2830 FOR N=1 TO 1000:NEXT N
2840 POKE 135,0
2850 C1$=INKEY$:IF C1$="" THEN 2
850
2860 PUT(0,0)-(10,10),PR,PSET
2870 GET(123,0)-(133,10),PR,G
2880 PUT(123,0)-(133,10),PR,PRES
ET
2890 C2$=INKEY$:IF C2$="" THEN 2
890
2900 PUT(123,0)-(133,10),PR,PSET
2910 GET(245,0)-(255,10),PR,G
2920 PUT(245,0)-(255,10),PR,PRES
ET
2930 PA$=INKEY$:IF PA$="" THEN 2
930
2940 IF PA$<>"Y" THEN PUT(245,0)
-(255,10),PR,PSET:GOTO 1000
2950 PAINT(X,Y),VAL(C1$),VAL(C2$
)
2960 PUT(245,0)-(255,10),PR,PSET
2970 PF=0:GOTO 1000
```

Arc

It is possible to form only an arc of a CIRCLE or ellipse if you define the start and end points, but first the Width and Height must be defined as described for ellipses.

```
4000 IF CF=0 THEN 1000
4010 PUT(XO-1,YO-1)-(XO+1,YO+1),
OC,PSET
4020 W=ABS(XO-X):H=ABS(YO-Y)
4030 FOR N=1 TO 1000:NEXT N
4040 POKE135,0
4050 GET(10,0)-(15,10),PR,G
4060 PUT(10,0)-(15,10),PR,PRESET
```

You can reach this routine from A but there is a practical problem in indicating the start and end points by single keys, as even the keys 0 to 9 would only allow 10 different arc points. The actual values for start and end points need to be between 0 and 1 and the following solution allows you to enter easily decimal numbers via INKEY$. First the start value. Line 4070 checks INKEY$ and if this is not empty then 4080 checks whether ST$ was CHR$(13) (= ENTER). If not ST$ is added onto the end of T1$ and another ST$ taken. Thus numbers are added onto T1$ until ENTER is pressed. In the same way the end point is built up in T2$ from FI$.

```
4070 ST$=INKEY$:IF ST$="" THEN 4
070
4080 IF ST$<>CHR$(13) THEN T1$=T
1$+ST$:GOTO 4070
4090 PUT(10,0)-(15,10),PR,PSET
4100 GET(230,0)-(245,10),PR,G
4110 PUT(230,0)-(245,10),PR,PRES
ET
4120 FI$=INKEY$:IF FI$="" THEN 4
120
4130 IF FI$<>CHR$(13) THEN T2$=T
2$+FI$:GOTO 4120
```

When both start and end points have been entered the arc is drawn.

```
4140 PUT(230,0)-(245,10),PR,PSET
4150 CIRCLE(XO,YO),W,,H/W,VAL(T1
$),VAL(T2$)
```

As you may want to add to this arc Q$ waits to see if you want A for arc again. If so T1$ and T2$ are emptied but the shape of the CIRCLE is retained as W and H are not reset.

```
4160 Q$=INKEY$:IF Q$="" THEN 416
0
4170 IF Q$="A" THEN T1$="":T2$="
":GOTO 4030
4180 CF=0:T1$="":T2$="":GOTO 100
0
```

SAVE/LOAD

Once you have completed your design then presumably you want to be able to Save it so S leads to the SAVE/LOAD routine which dumps the contents of the first four graphic pages to cassette as a machine code file which can be reloaded later.

```
3300 CLS:PRINT"SAVE OR LOAD":INP
UT Q$:IF LEFT$(Q$,1)="S" THEN 33
10 ELSE IF LEFT$(Q$,1)="L" THEN
3350 ELSE SCREEN 1,SN:GOTO 1000
3310 PRINT"SAVE":PRINT,,"FILENAM
E";:INPUT NA$
3320 CSAVEM NA$,1536,7679,6144
3330 SCREEN 1,SN:GOTO 1000
3350 PRINT"LOAD":PRINT,,"FILENAM
E";:INPUT NA$
3360 SCREEN 1,SN:CLOADM NA$
3370 GOTO 1000
```

Drawing with the joystick

Although you can move around the screen with the cursor keys it is sometimes more convenient to use this joystick routine which is called by J, as you can then make diagonal moves more easily. The cursor key routine is replaced by the joystick routine until one of the cursor keys is pressed again. The joystick is used to control direction rather than absolute position here (see earlier). The JOYSTK values for 0 and 1 are read into variables and a logic test against position limits used to update both X and Y.

```
2200 J0=JOYSTK(0):J1=JOYSTK(1):X
=X+(IN*((J0<=20)-(J0=>50))):Y=Y+
(IN*((J1<=20)-(J1=>50)))
```

After limit tests a replica of the usual cursor routine is used and then a test made to see if any key is being pressed.

```
2210 IF Y>YE THEN Y=YE ELSE IF Y
<YS THEN Y=YS
2220 IF X>XE THEN X=XE ELSE IF X
<XS THEN X=XS
2230 GET(X-1,Y-1)-(X+1,Y+1),CU,G
:PUT(X-1,Y-1)-(X+1,Y+1),CU,PRESE
T:FOR N=1 TO 10:NEXT:PUT(X-1,Y-1
)-(X+1,Y+1),CU,PSET
```

If any key is pressed PEEK(337) is less than 255 and the joystick routine is left.

```
2250 IF PEEK(337)<255 THEN 1000
```

If a key is not pressed then the joystick button is tested comparing PEEK(65280) with 126 and 254. If the button is not pressed a Blank Move is made but if the button is pressed a MOVE is made.

```
2260 IF PEEK(65280)<>254 AND PEE
K(65280)<>126 THEN DRAW"BM"+STR$
(X)+","+STR$(Y):GOTO 2200 ELSE D
RAW"C"+STR$(C1)+"M"+STR$(X)+","+
STR$(Y):GOTO 2200
2270 GOTO 1130
```

Entering character mode

The final single key command is @ which exits the drawing mode and goes to an alternative mode in which preformed characters are displayed. D$ is set to "0" so that these characters are initially drawn from left to right (see later).

```
2100 POKE 135,0:D$="0":GOTO 20
```

Character mode

Keycheck and cursor

The scale factor for DRAW (S) is set to four times the cursor increment, and INKEY$ tested. A cursor is formed by GET and PUT as before, but here it is a line instead of a box as the Y axis is of zero length. The flashing cursor is repeated until a key is pressed and if this key is @ the program goes back to the drawing mode.

```
20 S=IN*4:C$=INKEY$:GET(X,Y)-(X+
IN,Y),CU,G:PUT(X,Y)-(X+IN,Y),CU,
PRESET:FOR N=1 TO 10:NEXT N:PUT(
X,Y)-(X+IN,Y),CU,PSET:IF C$="" T
HEN 20 ELSE IF C$="@" THEN 1000
```

The ASCII value of the last key pressed is now calculated and used in a logic test against the arrow keys for cursor movement. A move of one standard character unit is made for each left and right cursor key movement, and a move of one and a half character units for each movement of the up and down keys. This gives the correct spacing between alphanumeric characters and lines. Once the X and Y limits have been checked a Blank Move is made to the new position.

```
21 A=ASC(C$):X=X+((S*2)*((A=8)-(
A=9))):Y=Y+((S*2)*((A=94)-(A=10)
)):IF Y>YE THEN Y=YE ELSE IF Y<Y
S THEN Y=YS
```

```
22 IF X>XE THEN X=XE ELSE IF X<X
S THEN X=XS
23 DRAW"BM"+STR$(X)+","+STR$(Y):
IF A>31 AND A<91 THEN GOSUB 25:X
=X+(S):GOTO 20:ELSE 20
```

If the ASCII value of the key is not between 32 and 90 the program loops back to 20, but if it is between these limits it goes to the subroutine at 25. On RETURN from this character DRAWing routine the X position is updated.

Sorting the characters

Line 25 is a key line as it sorts keys between 32 and 90 by an ON GOSUB according to the ASCII codes. Each of these subroutines DRAWs a different character and the program is arranged so that the line numbers of these subroutines correspond to the ASCII codes of the key pressed. For example pressing A leads to line 65. The only key with a code between 32 and 90 which does not have a subroutine is @ (code 64) as this key has already been used to return to the drawing mode.

```
25 DRAW"C"+STR$(C1)+"A"+D$+"S"+S
TR$(S):ON(ASC(C$)-31)GOSUB32,33,
34,35,36,37,38,39,40,41,42,43,44
,45,46,47,48,49,50,51,52,53,54,5
5,56,57,58,59,60,61,62,63,64,65,
66,67,68,69,70,71,72,73,74,75,76
,77,78,79,80,81,82,83,84,85,86,8
7,88,89,90:RETURN
```

Character subroutines

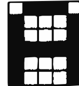
You can DRAW any type of character you like in the subroutines, the only proviso being that you must make sure that you finally make a Blank Move to a standard point in the next character position. The examples given **(Table 12.1)** include all the upper case letters and the numerals, together with some other special characters. The characters are constructed on a 5 by 6 grid **(Figure 12.2)**. If you want to define even more characters you can include lower case and duplicate line 25 as 26 with higher ASCII codes. The great advantage of using DRAW to produce characters is that these can be of any size and shape and be scaled, coloured and angled at will. Of particular interest are the accent routines which replace the normal characters on the #, $, % and & keys,

```
35 DRAW"BM-7,-7E3BM+4,+10":RETUR
N
36 DRAW"BM-4,-7H3BM+7,+10":RETUR
N
37 DRAW"BM-8,-7E2F2BM+4,+7":RETU
RN
38 DRAW"BM-6,+1DGBM+7,-3":RETURN
```

and the copyright sign which replaces the !

```
33 DRAW"BM+1,+0R3EU4HL3GD4FBM+2,
-2LHERBM+4,+4":RETURN
```

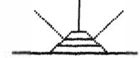
Figure 12.2 Letter A formed on 6 × 5 grid



The greater than (>) sign has been replaced by a larger design which should be reasonably familiar to Dragon users **(Figure 12.3).** As this is larger than the rest of the characters the X position is moved along further than usual.

```
62 DRAW"BM+2,+0R17BM--4,+0HL6GE2R
4FH2L2GH4BM+6,+3U5BM+2,+6E4":X=X
+(S*1.5):RETURN
```

Figure 12.3 An alternative character



Colour (C1), scale (S) and angle (D)

If you look again at the start of line 25 you will see that each character is DRAWn in the current foreground colour, at the current scale and angle. To change the foreground colour (C1) or scale (S) you must jump back to drawing mode and you may remember that D$ was set to "0" before

entering character mode, so that DRAWing proceeds from left to right. Four different scales are available and these produce different sizes of letters (**Figure 12.4**). Notice that line 32 sets the colour to C2 so that it DRAWs in background to produce both a space and a delete character feature.

Figure 12.4 Different letter sizes

The angle can be updated in character mode by a two-stage process. First a check is added for the ENTER key (CHR$(13)). If INKEY$ is not ENTER then line 24 is excluded. If ENTER is pressed INKEY$ is checked again and a warning SOUND made until another key is pressed. The 0 to 3 keys can now be used to change the angle of DRAWing. Invalid keys are rejected by looking at the VAL.

```
23 DRAW"BM"+STR$(X)+","+STR$(Y):
IF A=13 THEN 24 ELSE IF A>31 AND
 A<91 THEN GOSUB 25:X=X+(S*2):GO
TO 20 ELSE 20
24 D$=INKEY$:SOUND 1,1:IF D$=""
THEN 24 ELSE IF VAL(D$)>3 THEN D
$="":GOTO 20:ELSE 20
```

This feature is extremely useful in labelling diagrams as text can be written in all four directions (**Figure 12.5**).

Figure 12.5 Change of angle

Resurrection

If you are unfortunate enough to try the impossible and hence crash the program typlng SCREEN 1,SN:GOTO 1000 will usually put you back where you were before your last move.

Written prompts

Now that you have the facility to produce text on the hi-res screen you can easily change the 'block' prompts described earlier to actual written messages. The message to be given is defined as M$ and then this is sliced down one character at a time and sent through the normal character drawing routine. We will put the slicing routine at 6000 and set D$ = "0" to ensure that the angle command always gives normal text.

```
6000 D$="0":IN=2:FOR N=1 TO LEN(
M$):C$=MID$(M$,N,1):GOSUB 25:NEX
T N:RETURN
```

As a demonstration we will modify the PAINT routine. The easiest way to recreate the screen after the prompts is to GET the whole of the top of the screen into array WM.

```
2810 GET(0,0)-(255,10),WM,G
```

Now we define the first message as M$, make a Blank Move to the position you wish to write from, and go to the slicing subroutine.

```
2820 M$="COLOR 1?":DRAW "BM10,10
":GOSUB 6000
```

When the value has been entered as before the top is PUT back.

```
2860 PUT(0,0)-(255,10),WM,PSET
```

The other messages are dealt with in the same way, except that there is no need to GET into WM each time so certain lines can be deleted.

```
2870 (delete)
2880 M$="COLOR 2?":DRAW "BM50,10
":GOSUB 6000
2900 PUT(0,0)-(255,10),WM,PSET

2910 (delete)
2920 M$="PAINT Y/N?":DRAW "BM100
,10":GOSUB 6000
2940 IF PA$<>"Y" THEN PUT(0,0)-(
255,10),WM,PSET:GOTO 1000
2960 PUT(0,0)-(255,10),WM,PSET
```

Any other text message can be written onto the screen in the same way.

LIST OF SINGLE KEY COMMANDS

Key	Action
DRAWING MODE	move cursor
cursor keys	draw with cursor
shifted cursor keys	cursor increment 1
1	cursor increment 2
2	cursor increment 4
3	cursor increment 8
4	leave mark
spacebar	LINE from mark
L	Rubout LINE from mark
R	LINE no-update from mark
N	Box
B	Filled box
F	CIRCLE
C	ellipse

E GET

G backup copy

clear

P PUT PSET

I PUT PRESET

D PUT AND

O PUT OR

T PUT NOT

K kill (clear screen)
(must be held for 5 seconds)

W change colours
(first number is foreground,
second number is background)

Q PAINT
(Press twice, then enter paint colour,
border colour, and 'Y' if correct)

A arc

```
(first number is start, second number
 is end, press 'A' again to continue
 this arc)

S                             CSAVE/CLOAD

J                             JOYSTK
    (press a key to leave joystick mode)

@                             enter character mode

CHARACTER MODE

cursor keys                   move cursor

enter                         change angle
    (press 0-3 to select angle)

@                             enter drawing mode

any other key                 draw character


To recover from a crash type:

      SCREEN 1,SN:GOTO 1000
```

Table 12.1

SAMPLE CHARACTERS

```
32 DRAW"C"+STR$(C2)+"U6RD6RU6RD6
RD6RD6RBM+2,+0":RETURN
33 DRAW"BM+1,+0R3EU4HL3GD4FBM+2,
-2LHEREBM+4,+4":RETURN
34 DRAW"BM+0,-6DBM+2,+0UBM+4,+6"
:RETURN
35 DRAW"BM-7,-7E3BM+4,+10":RETUR
N
36 DRAW"BM-4,-7H3BM+7,+10":RETUR
N
37 DRAW"BM-8,-7E2F2BM+4,+7":RETU
RN
38 DRAW"BM-6,+1DGBM+7,-3":RETURN
39 DRAW"BM+0,-6DBM+4,+5":RETURN
40 DRAW"BM+2,+0HU4EBM+4,+6":RETU
RN
41 DRAW"BM+1,+0EU4HBM+5,+6":RETU
RN
42 DRAW"BM+0,-1E4BM+0,+4H4BM+8,+
5":RETURN
43 DRAW"BM+0,-3R4L2U2D4BM+5,+1":
RETURN
44 DRAW"BM-1,+0DGBM+4,-2":RETURN
45 DRAW"BM+0,-3R4BM+4,+3":RETURN
46 DRAW"BM-1,+0UBM+4,+1":RETURN
47 DRAW"BM+0,-1E4BM+4,+5":RETURN
48 DRAW"BM+0,-1FR2EU4HL2GD4BM+8,
+1":RETURN
49 DRAW"BM+1,+0U6GBM+6,+5":RETUR
N
50 DRAW"BM+4,+0L4UER2EU2HL2GBM+8
,+5":RETURN
51 DRAW"BM+0,-1FR2EUHL2R2EUHL2GB
M+8,+5":RETURN
52 DRAW"BM+3,+0U6G3R4BM+4,+3":RE
TURN
53 DRAW"BM+0,-1FR2EU2HL3U2R4BM+4
,+6":RETURN
54 DRAW"BM+0,-2ER2FDGL2HU4ER2FBM
+4,+5":RETURN
55 DRAW"BM+2,+0U2E2U2L4BM+8,+6":
RETURN
```

```
56 DRAW"BM+1,+0R2EUHL2HUER2FDGL2
GDFBM+7,+0":RETURN
57 DRAW"BM+0,-1FR2EU4HL2GDFR3BM+
4,+3":RETURN
58 DRAW"BM+0,-5DBM+0,+2DBM+4,+1"
:RETURN
59 DRAW"BM+0,-5DBM+0,+2DGBM+5,+0
":RETURN
60 RETURN
61 DRAW"BM+0,-2R4BM+0,-2L4BM+8,+
4":RETURN
62 DRAW"BM+2,+0R17BM-4,+0HL6GE2R
4FH2L2GH4BM+6,+3U5BM+2,+6E4":X=X
+(S*3):RETURN
63 DRAW"BM+2,+0UBM+0,-1UREUHLGBM
+7,+5":RETURN
65 DRAW"U5ER2FD5U3L4BM+8,+3":RET
URN
66 DRAW"U6R3FDGFDGL3U3R3BM+5,+3"
:RETURN
67 DRAW"BM+1,+0HU4ER2FHL2GD4FR2E
BM+4,+1":RETURN
68 DRAW"U6R3FD4GL3BM+8,+0":RETUR
N
69 DRAW"R4L4U3R4L4U3R4BM+4,+6":R
ETURN
70 DRAW"U3R4L4U3R4BM+4,+6":RETUR
N
71 DRAW"BM+1,+0R2EULRDGL2HU4ER2F
BM+4,+5":RETURN
72 DRAW"U6D3R4U3D6BM+4,+0":RETUR
N
73 DRAW"BM+1,+0R2LU6LR2BM+4,+6":
RETURN
74 DRAW"BM+0,-1FR2EU5BM+4,+6":RE
TURN
75 DRAW"U6BM+0,+3RE3G3F3BM+4,+0"
:RETURN
76 DRAW"R4L4U6BM+8,+6":RETURN
77 DRAW"U6F2E2D6BM+4,+0":RETURN
78 DRAW"U6DF4DU6BM+4,+6":RETURN
79 DRAW"BM+1,+0R2EU4HL2GD4FBM+7,
+0":RETURN
80 DRAW"U6R3FDGL3BM+8,+3":RETURN
81 DRAW"BM+1,+0R2EU4HL2GD4FBM+1,
```

```
-2F2BM+4,+0":RETURN
82 DRAW"U6R3FDGL3RF3BM+4,+0":RET
URN
83 DRAW"BM+0,-1FR2EH4ER2FBM+4,+5
":RETURN
84 DRAW"BM+2,+0U6L2R4BM+4,+6":RE
TURN
85 DRAW"BM+0,-6D5FR2EU5BM+4,+6":
RETURN
86 DRAW"BM+0,-6D4F2E2U4BM+4,+6":
RETURN
87 DRAW"BM+0,-6D6E2F2U6BM+4,+6":
RETURN
88 DRAW"UE4UBM+0,+6UH4UBM+8,+6":
RETURN
89 DRAW"BM+2,+0U4H2F2E2BM+4,+6":
RETURN
90 DRAW"R4L4UE4UL4BM+8,+6":RETUR
N
```

CHAPTER 13
GETting and PUTting Hi-Res Characters

Although the DRAW routines given above can be incorporated into any program they are somewhat slow in operation and you can quite easily 'beat' them if you type quickly. However, once you have drawn them on the screen you can save them and deal with them much faster later with GET and PUT.

Transferring characters between programs

Once you have created your characters it is obviously useful to be able to transfer these between programs, so that you do not have to retype them, but can build up a whole library of alternative sets instead. It is possible to define each character as a real string by changing each line to:

....A$ = "....."

instead of

....DRAW "...." etc

and then to save all these as an ASCII file on tape.

Saving characters as machine code

However it is actually much simpler to draw them across the top of the screen and then save this screen area as a machine code dump on tape. As an example we will set out the example letters and numbers which we defined earlier at the top of the screen and then save them. The screen is cleared and the colour set reversed to give black characters on a light background as these are easier to read. (If you want to save them as white on black just use PCLS instead of COLOR 0,1:PCLS1). We can set up the screen position by a Blank Move and then jump into the normal character drawing subroutine to lay out the letters and numbers in a suitable format.

```
10000 PMODE 4,1:SCREEN 1,0:COLOR
0,1:PCLS1:X=1:FOR C=48 TO 57:DRA
W"BM"+STR$(X)+",7":GOSUB 25:X=X+
7:NEXT C
```

```
10010 FOR C=65 TO 90:DRAW"BM"+STR
$(X)+",7":GOSUB 25:X=X+7:NEXT C
10020 CSAVEM "CHARS",1536,1722,22
4
```

The start position for the first character is coordinates 1,7, and each subsequent character is seven pixels to the right of this. C values of 48 to 57 define the numbers and 65 to 90 the letters.

In PMODE 4 there are 256/8 = 32 bytes on each line and we have only used the top seven lines so we need to CSAVEM 224 bytes to save all 36 characters. As that works out at less than seven bytes per character you can see that it is a very economical method, and you will also notice that machine code saves quickly.

Loading the characters and setting the screen

Characters which have been dumped on tape in machine code by CSAVEM as above can easily be recovered by CLOADM and used to produce a superior text display. For optimum visibility use PMODE 4 and reverse the COLOR to give black letters on a green (or buff) background.

```
10 PMODE 4,1:SCREEN1,0:PCLS1:COL
OR0,1:CLOADM
```

The characters will reappear in a single line across the top of the screen (**Figure 13.1**).

Figure 13.1 Reloaded characters

```
0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

Dimensioning the arrays

Before you GET all the characters you must DIM suitable arrays. Unfortunately that normally means a lot of repetitive typing as you cannot alter line numbers on the Dragon or switch the name of an array in GET and PUT. (In fact we have found a devious way of getting around that problem, but as it is not very easy to understand we have left it until later! If you want to become really good with graphics make sure you understand how this method works before trying the alternative.) The size of each array is only ONE element as the 5 by 7 matrix needs only 35 bits. Each array is named as C plus the actual character and in addition a blank array (BL) is also DIMensioned.

```
20 DIMC0(1):DIMC1(1):DIMC2(1):DI
MC3(1):DIMC4(1):DIMC5(1):DIMC6(1
```

```
):DIMC7(1):DIMC8(1):DIMC9(1):DIM
CA(1):DIMCB(1):DIMCC(1):DIMCD(1)
:DIMCE(1):DIMCF(1):DIMCG(1):DIMC
H(1):DIMCI(1):DIMCJ(1):DIMCK(1):
DIMCL(1):DIMCM(1):DIMCN(1)
30 DIM DU(1):DIM CO(1):DIMCP(1):
DIMCQ(1):DIMCR(1):DIMCS(1):DIMCT
(1):DIMCU(1):DIMCV(1):DIMCW(1):D
IMCX(1):DIMCY(1):DIMCZ(1):DIM BL
(1)
```

GETting the characters

Appropriate variables must now be set according to the size and spacing of the characters to be picked up. X and Y set the start coordinates, S is the step between characters on the screen, and W and H the actual Width and Height of the characters.

```
40 X=1:Y=0:S=7:W=5:H=7
```

All of the upper-case characters and the numbers can be fitted onto the top screen line so now you just need to GET each character, stepping the X coordinate by S each time.

```
100 GET(X,Y)-(X+W,Y+H),C0,G:X=X+
S
110 GET(X,Y)-(X+W,Y+H),C1,G:X=X+
S
120 GET(X,Y)-(X+W,Y+H),C2,G:X=X+
S
130 GET(X,Y)-(X+W,Y+H),C3,G:X=X+
S
140 GET(X,Y)-(X+W,Y+H),C4,G:X=X+
S
150 GET(X,Y)-(X+W,Y+H),C5,G:X=X+
S
160 GET(X,Y)-(X+W,Y+H),C6,G:X=X+
S
170 GET(X,Y)-(X+W,Y+H),C7,G:X=X+
S
180 GET(X,Y)-(X+W,Y+H),C8,G:X=X+
S
190 GET(X,Y)-(X+W,Y+H),C9,G:X=X+
S
200 GET(X,Y)-(X+W,Y+H),CA,G:X=X+
```

```
S
210 GET(X,Y)-(X+W,Y+H),CB,G:X=X+
S
220 GET(X,Y)-(X+W,Y+H),CC,G:X=X+
S
230 GET(X,Y)-(X+W,Y+H),CD,G:X=X+
S
240 GET(X,Y)-(X+W,Y+H),CE,G:X=X+
S
250 GET(X,Y)-(X+W,Y+H),CF,G:X=X+
S
260 GET(X,Y)-(X+W,Y+H),CG,G:X=X+
S
270 GET(X,Y)-(X+W,Y+H),CH,G:X=X+
S
280 GET(X,Y)-(X+W,Y+H),CI,G:X=X+
S
290 GET(X,Y)-(X+W,Y+H),CJ,G:X=X+
S
300 GET(X,Y)-(X+W,Y+H),CK,G:X=X+
S
310 GET(X,Y)-(X+W,Y+H),CL,G:X=X+
S
320 GET(X,Y)-(X+W,Y+H),CM,G:X=X+
S
330 GET(X,Y)-(X+W,Y+H),CN,G:X=X+
S
340 GET(X,Y)-(X+W,Y+H),CO,G:X=X+
S
350 GET(X,Y)-(X+W,Y+H),CP,G:X=X+
S
360 GET(X,Y)-(X+W,Y+H),CQ,G:X=X+
S
370 GET(X,Y)-(X+W,Y+H),CR,G:X=X+
S
380 GET(X,Y)-(X+W,Y+H),CS,G:X=X+
S
390 GET(X,Y)-(X+W,Y+H),CT,G:X=X+
S
400 GET(X,Y)-(X+W,Y+H),CU,G:X=X+
S
410 GET(X,Y)-(X+W,Y+H),CV,G:X=X+
S
420 GET(X,Y)-(X+W,Y+H),CW,G:X=X+
```

```
S
430 GET(X,Y)-(X+W,Y+H),CX,G:X=X+
S
440 GET(X,Y)-(X+W,Y+H),CY,G:X=X+
S
450 GET(X,Y)-(X+W,Y+H),CZ,G:X=X+
S
```

Setting the screen format

Once all the characters are safely in their arrays the screen can be cleared, and new variables set up to control the screen format. X and Y are the start coordinates, S is the step along the X axis, T the step along the Y axis, and XS, YS, XE and YE the limit coordinates values for X and Y axes. Although the actual size of the characters is fixed S controls the amount of space between characters on the X axis, and hence the number of characters per line. T controls the amount of space between lines of characters, and hence the number of lines which can be fitted on the screen. If only part of the screen is to be written on then the values of XS, YS, XE and YE must be modified. The combination of values given produces a matrix of 42 by 24 characters (a total of 1008 (almost double the number on the normal Dragon text screen). Although it takes several seconds to fill the arrays initially these are retained as long as you do not use RUN. If you crash the program make sure that you restart it by GOTO.

```
500 PCLS:X=2:Y=0:S=6:T=8:XS=2:XE
=253:YS=0:YE=191
```

Cursor and keycheck

K$ is read from INKEY$ and a flashing cursor produced, in this case by a double PUT of the BLank array with NOT to invert the screen twice. The first PUT, NOT reverses the state of all points in the area, and the second PUT, NOT reverses them again so they are back where they started. When a key is pressed the ASCII value is taken and if this falls above or below the code for the present characters the program jumps on to line 1500.

```
510 K$=INKEY$:PUT(X,Y)-(X+W,Y+H)
,BL,NOT:PUT(X,Y)-(X+W,Y+H),BL,NO
T:IF K$="" THEN 510 ELSE K=ASC(K
$):IF K<47 OR K>91 THEN 1500
```

The other keys are sorted by an ON GOSUB related to their codes.Six of the missing characters with codes between those of the numbers and upper-case letters jump back to the INKEY$ check, but @ goes to another routine at 1610.

```
520 ON K-47 GOSUB 1000,1010,1020
,1030,1040,1050,1060,1070,1080,1
090,510,510,510,510,510,510,1610
,1100,1110,1120,1130,1140,1150,1
160,1170,1180,1190,1200,1210,122
0,1230,1240,1250,1260,1270,1280,
1290,1300,1310,1320,1330,1340,13
50
```

PUTting the characters

There is a corresponding PUT, PSET subroutine for each character and the program then RETURNs. You can save yourself some typing if you CSAVE the program so far, delete everything except the GET lines, and then append your program back on itself. The procedure for appending is first to PEEK at locations 27 and 28 and then POKE location 25 with the number in 27 and location 26 with two less than the number in 28. This sets the 'start of BASIC program' pointer above the end of the program left in memory, and you can safely CLOAD your other COPY on top of this. You now RENUM it above the original lines and if you then POKE 25,30 and POKE 26,1 you will reset the start pointer back where it was and find both sets of lines form one program. You can now edit the copies of the GET lines to convert them into PUT lines.

```
1000 PUT(X,Y)-(X+W,Y+H),C0,PSET:
RETURN
1010 PUT(X,Y)-(X+W,Y+H),C1,PSET:
RETURN
1020 PUT(X,Y)-(X+W,Y+H),C2,PSET:
RETURN
1030 PUT(X,Y)-(X+W,Y+H),C3,PSET:
RETURN
1040 PUT(X,Y)-(X+W,Y+H),C4,PSET:
RETURN
1050 PUT(X,Y)-(X+W,Y+H),C5,PSET:
RETURN
1060 PUT(X,Y)-(X+W,Y+H),C6,PSET:
RETURN
1070 PUT(X,Y)-(X+W,Y+H),C7,PSET:
RETURN
1080 PUT(X,Y)-(X+W,Y+H),C8,PSET:
RETURN
```

```
1090 PUT(X,Y)-(X+W,Y+H),C9,PSET:
RETURN
1100 PUT(X,Y)-(X+W,Y+H),CA,PSET:
RETURN
1110 PUT(X,Y)-(X+W,Y+H),CB,PSET:
RETURN
1120 PUT(X,Y)-(X+W,Y+H),CC,PSET:
RETURN
1130 PUT(X,Y)-(X+W,Y+H),CD,PSET:
RETURN
1140 PUT(X,Y)-(X+W,Y+H),CE,PSET:
RETURN
1150 PUT(X,Y)-(X+W,Y+H),CF,PSET:
RETURN
1160 PUT(X,Y)-(X+W,Y+H),CG,PSET:
RETURN
1170 PUT(X,Y)-(X+W,Y+H),CH,PSET:
RETURN
1180 PUT(X,Y)-(X+W,Y+H),CI,PSET:
RETURN
1190 PUT(X,Y)-(X+W,Y+H),CJ,PSET:
RETURN
1200 PUT(X,Y)-(X+W,Y+H),CK,PSET:
RETURN
1210 PUT(X,Y)-(X+W,Y+H),CL,PSET:
RETURN
1220 PUT(X,Y)-(X+W,Y+H),CM,PSET:
RETURN
1230 PUT(X,Y)-(X+W,Y+H),CN,PSET:
RETURN
1240 PUT(X,Y)-(X+W,Y+H),CO,PSET:
RETURN
1250 PUT(X,Y)-(X+W,Y+H),CP,PSET:
RETURN
1260 PUT(X,Y)-(X+W,Y+H),CQ,PSET:
RETURN
1270 PUT(X,Y)-(X+W,Y+H),CR,PSET:
RETURN
1280 PUT(X,Y)-(X+W,Y+H),CS,PSET:
RETURN
1290 PUT(X,Y)-(X+W,Y+H),CT,PSET:
RETURN
1300 PUT(X,Y)-(X+W,Y+H),CU,PSET:
RETURN
```

```
1310 PUT(X,Y)-(X+W,Y+H),CV,PSET:
RETURN
1320 PUT(X,Y)-(X+W,Y+H),CW,PSET:
RETURN
1330 PUT(X,Y)-(X+W,Y+H),CX,PSET:
RETURN
1340 PUT(X,Y)-(X+W,Y+H),CY,PSET:
RETURN
1350 PUT(X,Y)-(X+W,Y+H),CZ,PSET:
RETURN
```

When you RUN this program you will no doubt be impressed by the speed of GET and PUT which appear to operate instantaneously, and certainly faster than you can type. This speed is the main advantage of this method of character generation, but of course this must be set against the impossibility of scaling and colouring the characters, or of changing their angle on the screen. It is chiefly a useful method of getting a reasonable amount of text on the screen at one time. Do not be tempted to save some typing by using GET without graphic detail and PUT without action as we have found that this theoretically desirable combination does not work very effectively in practice. There should be a speed advantage over the method described here, but in practice we have found that it actually tends to crash with mysterious FC ERRORs if you type fast.

Once a character has been PUT the X position is stepped on. If the limit of the X axis has been reached (X> XE) then the X coordinate is reset to the start position but the Y coordinate is moved down to the next line. If the end of the screen is reached suitable action must be taken.

```
540 X=X+S:IF X>XE THEN X=1:Y=Y+T
:IF Y>YE THEN 2000
550 GOTO 510
2000 STOP
```

Moving on

The spacebar (code 32) produces a blank move to the right, and this line also PUTs the BLank array back with PRESET at this point. This has the effect of erasing the screen at the current position so it is also used for deletion. Note that this array was never filled so that PRESET rather than PSET is appropriate.

```
1500 IF K=32 THEN PUT(X,Y)-(X+W,
Y+H),BL,PRESET:X=X+S:GOTO 510
```

The arrow keys can also be used to move around the screen in all four directions.

```
1510 IF K=8 THEN X=X-S:GOTO 1560
1520 IF K=9 THEN X=X+S:GOTO 1560
1530 IF K=94 THEN Y=Y-T:GOTO 156
0
1540 IF K=10 THEN Y=Y+T:GOTO 156
0
1550 GOTO 510
1560 IF X<XS THEN X=XS
1570 IF X>XE THEN X=XS:Y=Y+T
1580 IF Y<YS THEN Y=YS
1590 IF Y>YE THEN Y=Y-T
1600 GOTO 510
```

Another case

If you want true lower case you will have to load more characters and duplicate all the GETs and PUTs, but an alternative inverse case can be produced very easily (**Figure 13.2**). If you want to move into inverse characters just press @ which leads to line 1610 which toggles between the two cases by setting a flag (FL).

```
1610 IF FL=1 THEN FL=0:RETURN EL
SE FL=1:RETURN
```

Figure 13.2 42 × 24 display

```
THIS IS AN EXAMPLE OF TEXT PRODUCED ON THE
HIGH RESOLUTION SCREEN BY GETTING AND
PUTTING CHARACTERS STORED AS A MACHINE
CODE FILE OF THE GRAPHICS PAGES ON TAPE

012345678901234567890123456789012345678901

IN THIS PARTICULAR FORMAT THERE ARE
FORTYTWO CHARACTERS ON A LINE AND TWENTY
FOUR LINES ON THE SCREEN

INVERSE CHARACTERS CAN BE PRODUCED
```

Another line is now slipped in which PUTs the BLank array over the current character with NOT, thus inverting all the screen points.

```
530 IF FL=1 THEN PUT(X,Y)-(X+W,Y
+H+1),BL,NOT
```

More or less characters

The 42 by 24 character display described is the largest that can be comfortably dealt with (**Figure 13.2**). The space between characters can be reduced by dropping S to 5, which gives 51 characters per line (1224 per screen) but this is pushing things very close to the limit (**Figure 13.3**). Increasing S to 7 reduces the number of characters per line to 36 (**Figure 13.4**) and if S is 8 you are back with the usual 32 characters per line of the Dragon text screen, although as there are still 24 lines you have 768 characters on the screen rather than 512. There must obviously be a compromise between legibility and quantity and with this technique you can make your choice according to external factors.

Figure 13.3 51 × 24 display

```
012345678901234567890123456789012345678901234567890

IN THE EXTREME CASE A MAXIMUM OF FIFTYONE
CHARACTERS CAN BE FITTED ONTO THE SCREEN ALTHOUGH
LEGIBILITY SUFFERS SOMEWHAT IF THE STEP SIZE IS
REDUCED THIS FAR
```

Figure 13.4 36 × 24 display

```
012345678901234567890123456789012345

EVEN THIS WIDELY SPACED 36 COLUMN
FORMAT HAS 24 LINES ON A SCREEN AND
THEREFORE 864 CHARACTERS PER SCREEN
INSTEAD OF THE NORMAL 512 ON THE
[illegible] TEXT SCREEN
```

CHAPTER 14
Working on a Grid

Although it is possible to build up freehand designs on the screen some form of grid system gives a very useful guide when you want to make sure that your figure fits a particular format. This is very important when you want to define character sets or frames for animation. The grid system described here is a much more powerful derivative of the paper 'plotting chart' idea that makes rubbers and Tippex redundant.

Making your choice

Before we can build up a grid on the screen we must decide what PMODE and colours to use, and what size, shape and scale the grid will be.

```
20 CLS:PRINT"PMODE";:INPUT PM:PR
INT"COLOR SET";:INPUT CS:PRINT"F
OREGROUND";INPUT Q1:PRINT"BACKRO
UND";INPUT Q2:CLS:PRINT"GRID WID
TH";:INPUT W:PRINT"GRID HEIGHT";
:INPUT H:PRINT"SCALE";:INPUT SX
```

Forming the grid

Now to set up the screen and define the start position (XS, YS) of the grid.

```
30 PMODE PM,1:SCREEN 1,CS:PCLS Q
2
40 XS=10:YS=50:SY=SX
```

XS = 10, YS = 50 starts the grid about one quarter of the way down the left-hand side of the screen, slightly away from the left edge. For each grid element to be square the scale factor for the Y axis (SY) must be the same as the scale factor for the X axis (XS) which you entered.

The end coordinates for the grid (XE, YE) are calculated by multiplying the width of the grid requested (W) by the X axis scale factor (SX), and the height of the grid requested (H) by the Y axis scale factor (SY). A check must then be made to ensure that the calculated area will fit on the screen.

If this check fails the program RUNs again. The current screen position (XP,YP) is set to the grid start (XS,YS).

```
50 XE=XS+(SX*W):YE=YS+(SY*H):XP=
XS:YP=YS:IF XE>190 OR YE>180 THE
N RUN
```

The specified limit of 190 for XE leaves a clear area to the right of the screen. For test purposes choose PMODE 4,1, foreground 1, background 0, width 10, height 10, scale 10.

The actual grid can now be drawn by a series of LINEs between start and end, spaced the scale factor apart.

```
60 COLOR Q1,Q2:N=YS:FOR M=XS TO
 XE STEP SX:LINE(M,N)-(M,N+(YE-
YS)),PSET:NEXT M:M=XS:FOR N=YS
TO YE STEP SY:LINE(M,N)-(M+(XE-
XS),N),PSET:NEXT N
```

Flashing cursor

You need a cursor to indicate your position in the array, and as this is formed by PUTting a blank array (B) this must be dimensioned first.

```
10 DIM B(10)
```

The actual flashing cursor is formed by PUTting the blank array at the current screen position twice with NOT. The first NOT inverts that sector of the grid and the second inverts it again to produce the original display.

```
80 FL=5:FOR R=1 TO 2:PUT(XP,YP)-
(XP+SX,YP+SY),B,NOT:FOR T=1 TO R
^FL:NEXT T:NEXT R:IF PEEK(337)=2
55 THEN 80 ELSE A=PEEK(135)
```

As long as no key is pressed (PEEK(337) = 255) this sequence repeats. To make the actual state of the grid sector under the cursor easier to see there is a timing loop which is related to whether this is the first (R = 1) or second (R = 2) NOT. The rate of flashing is also linked to a variable (FL) which has profound effects as the time delay calculation is exponential (R↑FL). A value of 5 for FL produces a reasonable effect. When a key is pressed the value of PEEK(135) is read into A.

Moving round the grid

The cursor keys control movement, and limits are tested so that it is not possible to leave the grid.

```
160 XI=((A=8)-(A=9))
170 YI=((A=94)-(A=10))
180 XP=XP+(XI*SX):YP=YP+(YI*SY)
190 IF XP>XE-SX THEN XP=XE-SX:GO
TO 80 ELSE IF XP<XS THEN XP=XS:G
OTO 80
200 IF YP>YE-SY THEN YP=YP-SY:GO
TO 80 ELSE IF YP<YS THEN YP=YS:G
OTO80
210 GOTO 80
```

Notethat thesize of the move is related to the scale factors (SX and SY).

Filling the grid and changing your mind

To fill in sectors of the grid we use another array (W) which is originally filled with the contents of the screen at the start coordinates.

```
10 DIM W(10):DIM B(10)
70 GET(XS,YS)-(XS+SX,YS+SY),W,G:
POKE 135,0
```

(The POKE 135,0 at the end is to cancel autorepeat when a redraw of the grid is called later.)

The key code for filling (32) is the spacebar, which was chosen as this is the most frequent request.

```
100 IF A=32 THEN PUT(XP,YP)-(XP+
SX,YP+SY),W,PRESET:GOTO 80
```

As W is PUT back PRESET it inverts the display. The program then loops back tothe cursor routine.

To remove a filled block from the grid the same array is PUT, PSET if "X" is pressed.

```
110 IF A=88 THEN PUT(XP,YP)-(XP+
SX,YP+SY),W,PSET:GOTO 80
```

Making a 'real' copy

The most valuable applications of this program are creation of characters and animation frames, so we need to be able to transfer our ideas from the grid to the actual screen. This can be done by PSET and PRESET of appro-

priate points and a miniature copy produced to the right of the grid from coordinates CX,CY.

```
40 XS=10:YS=50:SY=SX:FL=5:CX=200
:CY=90:XC=CX:YC=CY
```

The current position is XC,YC and this is only updated if the move was within the grid as line 210 is only reached after valid moves.

```
210 XC=XC+XI:YC=YC+YI:GOTO 80
```

PSET and PRESET of XC,YC are added to the previous fill and erase lines.

```
100 IF A=32 THEN PUT(XP,YP)-(XP+
SX,YP+SY),W,PRESET:PSET(XC,YC):G
OTO 80
110 IF A=88 THEN PUT(XP,YP)-(XP+
SX,YP+SY),W,PSET:PRESET(XC,YC):G
OTO 80
```

If you RUN this again you will see that all your actions on the grid are now mirrored in a smaller version to the right of the screen.

Storing the copies

When you have built up a satisfactory copy you can store it at the top of the screen by pressing "@". This GETs the copy at the right of the screen into array CH and PUTs this back in the top quarter.

```
10 DIM W(10):DIM B(10):DIM CH(50
0)
40 XS=10:YS=50:SY=SX:FL=5:CX=200
:CY=90:XC=CX:YC=CY:C1=0:C2=0
120 IF A=64 THEN GET(CX,CY)-(CX+
W,CY+H),CH,G:PUT(C1,C2)-(C1+W,C2
+H),CH,PSET:C1=C1+W:GOTO 60
```

The initial PUT coordinates are predefined as C1, C2 and the X axis position (C1) is moved across by the number of units in the width of the grid (W) after each PUT.

As this routine returns to line 60 it redraws the grid but does not clear it. This is very useful if you want to make a series of frames for animation (see later). When the grid is redrawn the gaps between blocks disappear so that the status is obvious.

Starting again

Should you decide that you do not like the contents of your grid, and want to wipe it clean, pressing CLEAR will give a partial screen clearance.

```
130 IF A=12 THEN FOR P=2 TO 4:PM
ODE 0,P:PCLS Q2:NEXT P:PMODE PM,
1:GOTO 60
```

Only pages 2 to 4 are cleared as the PCLS is done in PMODE 0. This command is also used if you want to clear a grid after storing a copy at the top of the screen. As these copies are on page 1 they are not affected by the CLEAR routine. Should you really want to destroy the stored copies press 3 to clear page 1.

```
140 IF A=51 THEN PMODE0,1:PCLS Q
2:PMODE 4,1:POKE 135,0:GOTO 60
```

If you decide that even the grid size is wrong then press "1" to RUN the program again.

```
90 IF A=49 THEN RUN
```

Inversion

A partial inversion of the screen in both grid and copy areas is possible. The copy area has the CH array PUT, NOT over it, whilst the grid is reversed by repeatedly PUTting array W, NOT, whilst moving down the appropriate screen area. If you repeat the action in this way you can use a smaller array than otherwise necessary.

```
150 IF A=73 THEN FOR N=YS-SY TO
YE+SY:PUT(XS-SX,N)-(XE+SX,N),W,N
OT:NEXT N:PUT (CX,CY)-(CX+SX,CY+
SY),CH,NOT:GOTO 80
```

This allows you to view and store an inverted copy but the screen should be cleared before continuing.

Saving

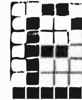
When you have formed all the characters or frames you need you can CSAVEM the screen as described previously.

Applications

Any kind of figure can be created on this system, and the grid may be small or large. In fact this routine has been used to provide material for several other chapters. Text characters are easily defined, for example **Figure 14.1** shows a set of true lower case in store with a pound sign about to be added. A Dragon logo is formed in **Figure 14.2**, and a tractor in **Figure 14.3**. We will leave you to add the rest of the farm implements. Production for a series of frames for animation is described elsewhere.

Figure 14.1 Generation of lower case and special characters

abcdefghijklmnopqrstuvwxyz



£

Figure 14.2 Dragon logo

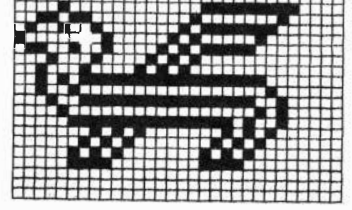
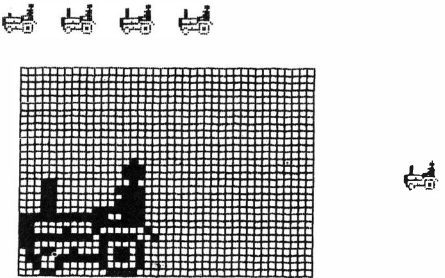


Figure 14.3 Tractor

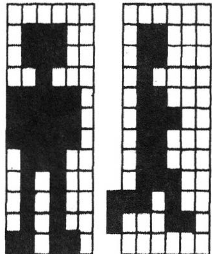


CHAPTER 15
Animation

Runner

The next step on from simply moving things around the screen is to animate a design — that is move parts of it to give the impression that it is alive. Once again we'll look at a SET point approach and consider producing the effect of a figure running. First we designed two alternative figures, the first showing a stationary figure facing forwards, and the second a running figure facing to the right (**Figure 15.1**).

Figure 15.1 Runner



The coordinates are in DATA statements and READ into arrays as before, but in this case there are really two alternative sets of coordinates. The first 35 points (in line 5020) make up one figure (stationary), and points 36 to 59 (in line 5030) the other (running). Only two arrays need be used as we can take any points from an array at any time and do not have to start from the beginning of the DATA each time. Separate subroutines to SET each figure are in lines 1000 and 2000.

If you trace the order of the DATA points and then watch this program in operation you will see that the simple animation effect is achieved because the leg points are SET relatively slowly and in a particular

sequence so that one leg appears before the other. There is no point in converting these SET points to the equivalent CHR$ as the increase in speed would mask the effect of movement here.

The sequence of operation is as follows. The title is printed and if no key is pressed then the first figure is displayed by the subroutine at 1000. If a key is pressed the program drops through to 120 which updates the screen offset (XO), clears the old picture, goes to the subroutine to SET the second figure (2000), clears to screen again, and reprints the title.

```
10 GOSUB 5000
20 CLS0
30 XO=2:YO=0:C=2
100 PRINT @256,"RUNNER"
110 IF PEEK(337)=255 THEN GOSUB
1000:GOTO 110
120 XO=XO+1:CLS0:GOSUB 2000:CLS0
:GOTO 100
1000 FOR N=1 TO 35:SET(X(N)+XO,Y
(N)+YO,C):NEXT N:RETURN
2000 FOR N=36 TO 59:SET(X(N)+X0,
Y(N)+YO,C):NEXT N:SOUND1,1:RETUR
N
5000 DIM X(59),Y(59)
5010 FOR N=1 TO 59:READ X(N),Y(N
):NEXT N:RETURN
5020 DATA 1;1,2,1,3,1,1,2,2,2,3,
2,2,3,0,4,1,4,2,4,3,4,4,4,0,5,1,
5,2,5,3,5,4,5,0,6,1,6,2,6,3,6,4,
6,1,7,2,7,3,7,1,8,3,8,1,9,3,9,1,
10,3,10,0,11,1,11,3,11,4,11
5030 DATA 1,1,1,2,2,1,2,2,1,3,1,
4,1,5,1,6,1,7,2,4,2,5,2,6,2,7,3,
5,1,8,2,8,3,8,3,9,3,10,4,10,1,9,
0,9,-1,9,-1,10
```

An alternative to CLS0 is to use a single 192 character string (BL$) to erase only the top of the screen (PRINT positions 0 to 191).

```
40 BL$=STRING$(192,128)
120 XO=XO+1:PRINT @0,BL$;:GOSUB
2000:PRINT @0,BL$;:GOTO 110
```

Sprinter

The runner described above appeared to move because of the slowness of SET and RESET and it is also possible to use the techniques described for

these with PSET and PRESET in hi-res. However you can produce much smoother animation in hi-res if you use GET and PUT, although of course you still need to make the pictures to GET and PUT first. **Figures 15.2** and **15.3** show two 'frames' of the movement of a sprinter which can be formed by PSETting the coordinates given in the DATA statements.

```
10 DATA 4,0,5,0,6,0,7,0,4,1,5,1,
6,1,7,1,4,2,5,2,6,2,7,2,4,3,5,3,
6,3,7,3,4,4,5,4,6,4,7,4,5,5,6,5,
4,6,5,6,6,6,7,6,4,7,5,7,7,7,3,8,
4,8,7,8,2,9,3,9,4,9,7,9,2,10,4,1
0,7,10,8,10,9,10,10,10,11,10,1,1
1,2,11,4,11,7,11,8,11,9,11,10,11
,11,11,12,11,2,12,3,12
20 DATA 4,12,5,12,6,12,7,12,3,13
,4,13,5,13,6,13,7,13,4,14,7,14,4
,15,5,15,6,15,7,15,4,16,5,16,6,1
6,7,16,4,17,5,17,6,17,7,17,8,17,
9,17,10,17,4,18,5,18,6,18,7,18,8
,18,9,18,10,18,5,19,6,19,10,19,1
1,19,5,20,6,20,10,20,11,20,5,21,
6,21,10,21,11,21,5,22,6,2
30 DATA 10,22,11,22,12,22,13,22,
5,23,6,23,10,23,11,23,12,23,13,2
3,5,24,6,24,5,25,6,25,5,26,6,26,
7,26,8,26,5,27,6,27,7,27,8,27
40 DATA 55,0,56,0,57,0,58,0,55,1
,56,1,57,1,58,1,55,2,56,2,57,2,5
8,2,55,3,56,3,57,3,58,3,55,4,56,
4,57,4,58,4,56,5,57,5,55,6,56,6,
57,6,58,6,55,7,57,7,58,7,54,8,55
,8,57,8,58,8,53,9,54,9,55,9,57,9
,58,9,53,10,55,10,57,10,58,10,59
,10,60,10,61,10,62,10
50 DATA 52,11,53,11,55,11,57,11,
58,11,59,11,60,11,61,11,62,11,63
,11,53,12,54,12,55,12,58,12,54,1
3,55,13,58,13,55,14,58,14,55,15,
56,15,57,15,58,15,55,16,56,16,57
,16,58,16,55,17,56,17,57,17,58,1
7,55,18,56,18,57,18,58,18,59,18,
56,19,57,19,59,19,60,19
60 DATA 56,20,57,20,60,20,61,20,
56,21,57,21,61,21,62,21,51,22,52
```

```
,22,53,22,54,22,55,22,56,22,57,2
2,61,22,62,22,51,23,52,23,53,23,
54,23,55,23,56,23,57,23,61,23,62
,23,51,24,52,24,61,24,62,24,51,2
5,52,25,61,25,62,25,63,25,64,25,
61,26,62,26,63,26,64,26
80 PMODE 4,1:SCREEN1,0:PCLS
90 FOR N=1 TO 249
100 READ X,Y:PSET(X+10,Y)
110 NEXT N
```

Figure 15.2 Sprinter Frame 1

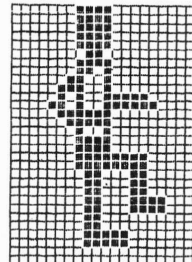
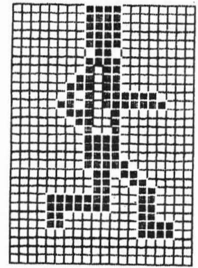


Figure 15.3 Sprinter Frame 2



Once the two frames have been PSET (a very slow job, but at least it only has to be done once!) you can GET them into arrays F1 and F2, and PCLS away the figures you just PSET ready for the animated sequence.

```
70 DIM F1(50):DIM F2(50)
120 GET(5,0)-(30,27),F1,G
130 GET(55,0)-(80,27),F2,G
140 PCLS
```

The simplest sequence is to PSET each array in turn so that the figure runs on the spot half way down the left hand side of the screen.

```
180 PUT(X,100)-(X+25,127),F1,PSE
T
200 PUT(X,100)-(X+25,127),F2,PSE
T
230 GOTO 180
```

If you now arrange to increment X in a FOR. . .NEXT loop he runs across the screen from left to right. Notice that each frame is shown at each X step before X is updated. **Figure 15.4** shows the two frames frozen alternately at a number of screen positions.

```
160 FOR X=1 TO 230  STEP 5
220 NEXT X
230 GOTO 140
```

Figure 15.4 Frozen sprinters

He moves very smoothly and quite rapidly across the screen, but what happens if there is a visible background behind him? Add in some horizontal lines to give a test background and RUN again.

```
150 FOR LI=1 TO 30 STEP 5:LINE(0
,LI)-(255,LI),PSET:NEXT LI
```

As you can see from **Figure 15.5** the lines vanish as the man runs over them, which is not much use in a real program. Now we could GET the background just before we PUT the figure and then PUT the background back when it moved on. We only GET the background once for both

frames as it is the original background we need to PUT back. If we just recreate the background with PSET then the lines reappear (**Figure 15.6**) but there is a lot of flashing and whilst each frame of the figure is shown part of the lines are erased.

```
70 DIM F1(20):DIM F2(20):DIM BG(
20)
170 GET(X,0)-(X+25,27),BG,G
210 PUT(X,0)-(X+25,27),BG,PSET
```

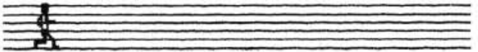
Figure 15.5 Background erased

Figure 15.6 PUT, PSET figure over background

To get a smoother replacement of the background (**Figure 15.7**) we need to make things a little more complicated, and apply some logical actions in our PUT commands.

```
180 PUT(X,0)-(X+25,27),F1,OR
190 PUT(X,0)-(X+25,27),BG,AND
200 PUT(X,0)-(X+25,27),F2,OR
210 PUT(X,0)-(X+25,27),BG,AND
```

Figure 15.7 The logical answer



First we PUT the first frame (F1) over the background (BG) with OR. This gives background plus frame 1 as all points which are set in either array OR screen are set. Now we PUT back the background (BG) with AND so that only points which are common to both the current screen and the original screen remain set. This produces the original position and we can then PUT the second frame (F2) with OR and then AND this with the

background (BG) as for frame 1. Notice that it is essential to PUT the background back between frames if you are to avoid problems with the International Athletics Association over three-legged sprinters.

Flying high

The degree of realism in an animation depends on the accuracy of the frames, but also on the number of frames in the sequence. As an example **Figures 15a–e** show a series of four different pictures of a bird in flight. The easiest way to produce such a series of related frames is to generate them on the screen grid system described earlier. The first picture shows the wings in the highest position and when @ is pressed to transfer the finished design to the top of the screen **(Figure 15.8b)** the grid lines are redrawn so that the points blocked in now form a solid pattern. This feature makes it simpler to construct the design of each subsequent frame as you can easily modify the existing picture, but still see which parts have been changed. When all the frames have been finished you can CSAVEM the top of the screen and on reloading GET and PUT these designs without ever actually thinking about which points you have PSET. Obviously that is a lot easier than typing in long DATA statements, but it does mean that you will have to copy our pictures onto the screen grid instead if you want this bird to fly. By this time you should be able to modify the sprinter program above to GET and PUT the correct areas.

Figure 15.8 Flying high

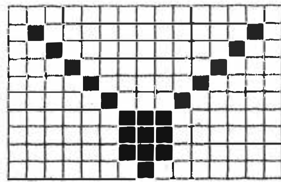


Figure 15.8b

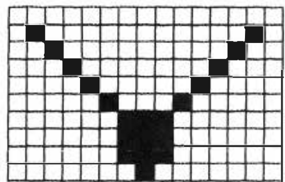
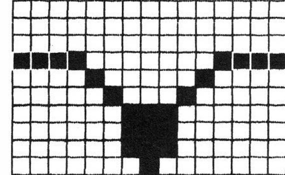


Figure 15.8c



Oasis

The only real disadvantages of GET and PUT animation are that you cannot change the scale, colour or angle of your design. DRAW will allow you to change these factors, but as it is slower than GET and PUT it is only useful for some applications, and new designs are best produced on graphics pages which are out of sight and then PCOPYed back to the current screen. As an example we will look at producing an oasis in the desert which gets bigger as you approach it.

First we need to PCLEAR all eight graphics pages and PCLS the first four to yellow (colour 2) to represent the sand.

```
10 PCLEAR 8:PMODE 3,1:SCREEN 1,0
:PCLS2
```

Figure 15.8d

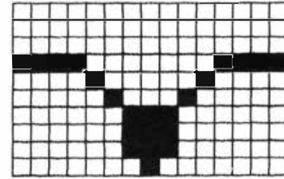


Figure 15.8e

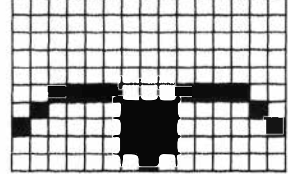
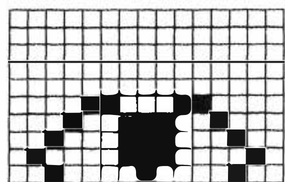


Figure 15.8f



The quickest way to set the top half of the screen to blue for the sky is to change the PMODE to 1 (which only uses two pages) and PCLS to 3. Remember that as there is no SCREEN command you are still looking at PMODE 3. Now we change the PMODE back to 3 and make a painted circle for the sun.

```
20 PMODE 1,1:PCLS 3:PMODE 3,1:CI
RCLE(230,30),20,2:PAINT(230,30),
2,2
```

In each picture the oasis is built up on a hidden screen on pages 5 to 8. PCLS 2 in PMODE 3 sets this to yellow and then PCLS 3 in PMODE 1 makes the top half blue.

```
50 PMODE 3,5:PCLS2:PMODE 1,5:PCL
S3:PMODE 3,5
```

The actual oasis is produced with DRAW and PAINT.

```
40 A$="C1LGER2FHLFHGEC4":W$="BM1
28,110C3BM-6,+0FR10EL13BM+6,+0":
PT$="C4U5XA$;BM+4,+5U4XA$;BM-8,+
4U3C1XA$;"
60 DRAW W$:PAINT(128,111),3,3:DR
AW PT$
```

To see the oasis on the screen we must PCOPY the last three pages of the hidden screen onto the last three pages of the screen display. As the top page does not change there is no point copying this.

```
70 PCOPY 6 TO 2:PCOPY 7 TO 3:PCO
PY 8 TO 4
80 SOUND(S*5),1
90 GOTO 90
```

If you RUN this you will see a minute oasis in the far distance (**Figure 15.9**), but if you add an incrementing scale factor (S) it will increase rapidly in size.

```
30 FOR S=4 TO 48 STEP 4:DRAW"S"+
STR$(S)
90 NEXT S
```

Figure 15.9a Oasis (scaled down animation)

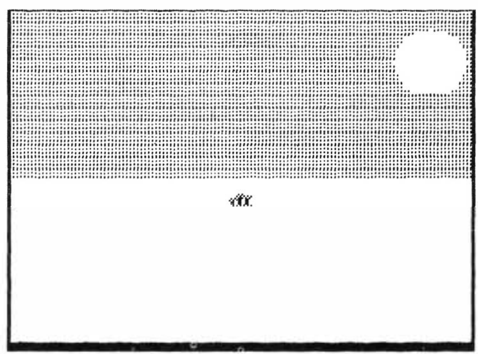


Figure 15.9b

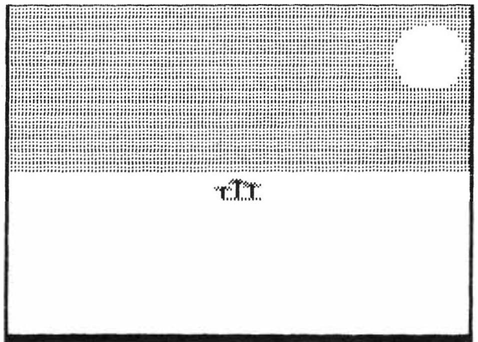


Figure 15.9c

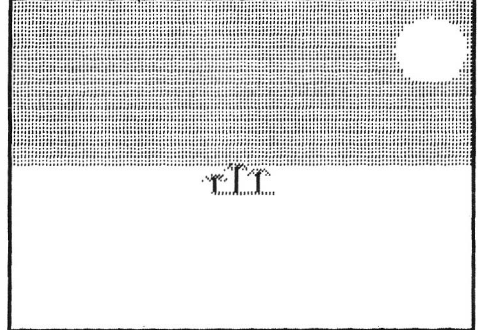


Figure 15.9d

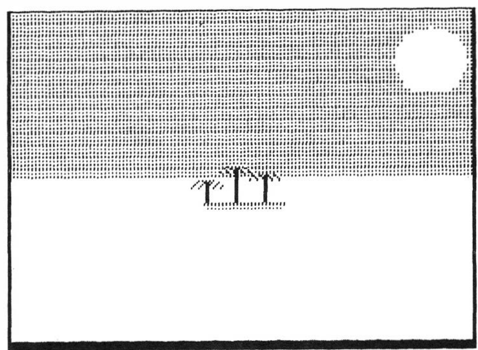


Figure 15.9e

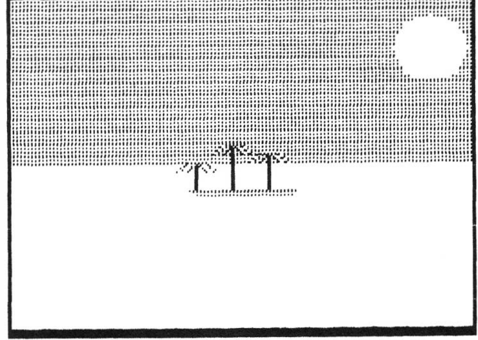


Figure 15.9f

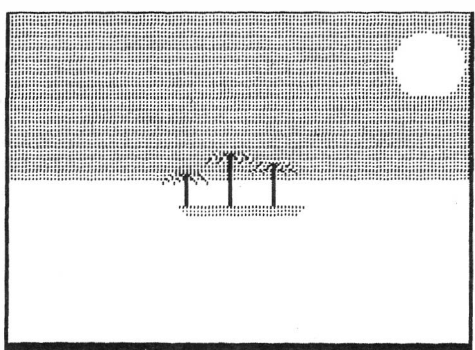


Figure 15.9g

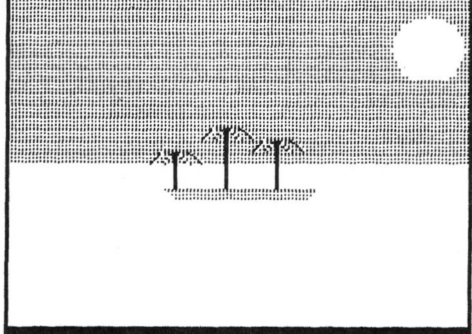


Figure 15.9h

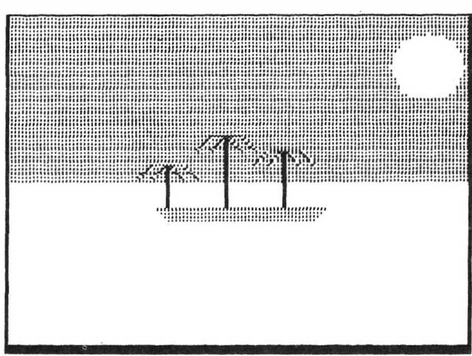


Figure 15.9i

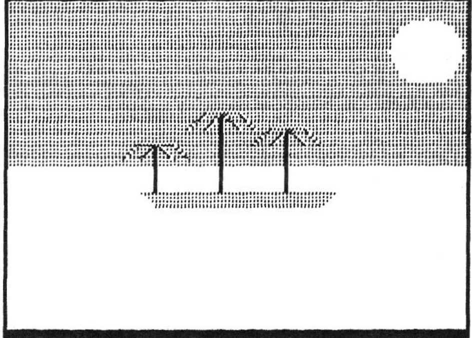


Figure 15.9j

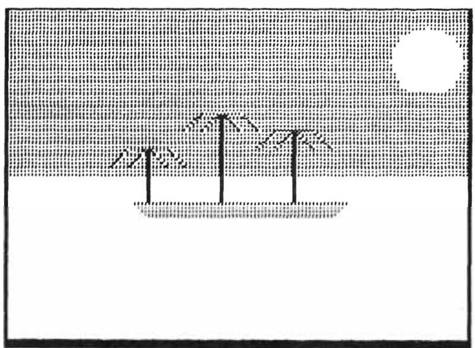


Figure 15.9k

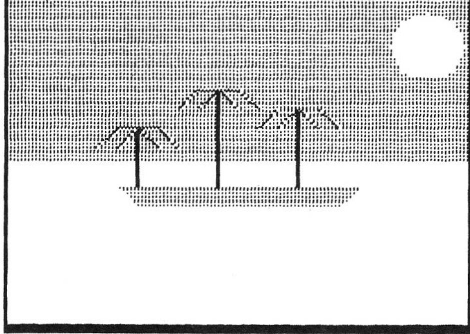
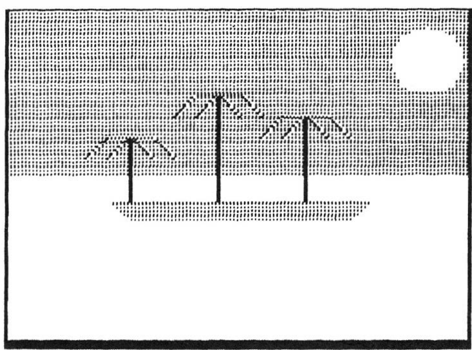


Figure 15.91



Of course mirages are very common in the desert so you shouldn't be too surprised when your head starts to spin and the oasis vanishes into the distance again.

```
100 FOR A=0 TO 3:PMODE 3,1:PCLS2:
PMODE 1,5:PCLS3:PMODE 3,5:DRAW "
A"+STR$(A)+"S"+STR$(48-(16*A))+W
$+PT$
110 PCOPY 6 TO 2:PCOPY 7 TO 3:PC
OPY 8 TO 4:NEXT A
120 FOR N=255 TO 1 STEP -5:SOUND
 N,1:NEXT N
130 RUN
```

PEEKING

```
10  CLS
20  X = 0
30  PRINT @ 100, X;
40  A = PEEK (135)
50  A$ = CHR$(A)
60  IF PEEK (33) = 255 THEN 100
70  IF A$ = CHR$(32) THEN X = X+1
80  IF A$ = CHR$(10) THEN X = X-1
100 GOTO 30
```

CHAPTER 16
Sound Synthesis

Although you are never going to get your Dragon to sound like a real synthesiser it is possible to demonstrate some of their features with this program. (It is generally known as the "MOG" synthesiser, as the results are rather feline at times).

Repeating keyboard sound

Although we showed you earlier how to PLAY notes directly from the keyboard with INKEY$ this method had two main disadvantages. The first is the fact that INKEY$ does not autorepeat so you have to lift your finger from the key before you can sound a note again. The second problem is that the keys for C D E F G A B are not in very logical places for playing music! As you have already seen making keys autorepeat is easy if you PEEK at location 135 which contains the ASCII code of the last key pressed. If we convert this to its character representation which is CHR$ we can PLAY it and a loop will then sustain the note until the key is released. (Notice that it is not a case of converting the number to a string with STR$ but of making the character with CHR$.)

```
80 A=PEEK(135)
130 A$=CHR$(A)
180 PLAY A$
190 GOTO 80
```

If you try that routine you will find that it doesn't work, although not because the logic is wrong! First of all there is a problem with the keybounce of the ENTER key you used to RUN the program, and secondly location 135 retains the code for the last key pressed. A preliminary pause gets over the keybounce, and a PEEK at 337 will tell us if a key has been pressed. Add these lines and keys A–G will autorepeat satisfactorily.

```
60 FOR N=1 TO 100:NEXT N
70 IF PEEK(337)=255 THEN 110
190 GOTO 70
```

The autorepeat is rather slow so change the default tempo setting to speed things up. A value of around T50 gives a reasonable effect.

```
50 PLAY"T50"
```

At the moment the note stops as soon as you release the key, but if you jump back to 120 instead of 110 the last note will continue to be repeated until you press another key.

```
190 GOTO 80
```

Reconfiguring the keyboard

Pressing an invalid key will still cause an FC ERROR but now that we can repeat a note continuously, or alternatively permanently sustain it, let's think about rearranging the keyboard. If you compare the Dragon key layout to a piano keyboard you can see that a more suitable set of keys to use would be Z S X D C V G B H N J and M **(Figure 16.1)**, where Z X C V B N and M represent the white notes and S D G H and J the black notes. A simple way to convert these keys to suitable notes using the INSTR function. First we need to set up a string (K$) containing all the valid keys. Next we need to compare the key pressed with K$. If the key pressed does not correspond with a character in K$ then INSTR will give 0. If, on the other hand, there is a match with a character in K$ then INSTR will give a number corresponding to the position of that character in the string.

```
10 K$="ZSXDCVGBHNJM"
140 C=INSTR(1,K$,A$)
150 IF C>0 THEN A$=STR$(C) ELSE
80
```

When you RUN this you will find that the designated notes function as the scale CDEFGAB and any other notes are ignored. But how was this miracle achieved? Well to understand that you must go back to square one again. Although so far we have only considered PLAYing notes designated by letters the Dragon also understands that the numbers 1 to 12 represent the same notes **(Figure 16.2)**. What we have done is arrange the notes in K$ in such an order that their position actually gives the corresponding number.

A second octave

If we set up another string containing a different set of characters **(Figure 16.3)** we can use these to produce a higher octave if we add "0 + " before the note, but we must then put "0 – " after the note to reset the octave.

```
20 L$="T6Y7UI9O0P:@"
150 IF C>0 THEN A$=STR$(C):GOTO
180
160 C=INSTR(1,L$,A$)
170 IF C>0 THEN A$="O+"+STR$(C)+
"O-": ELSE 80
```

Figure 16.1 Reconfiguration of keyboard

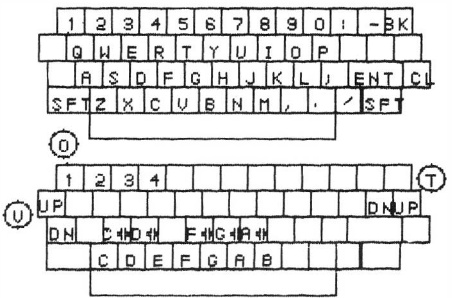
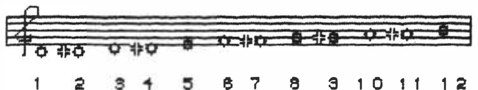


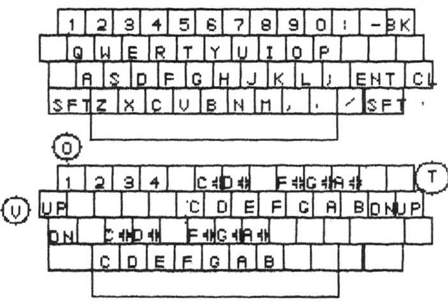
Figure 16.2 Number representation of notes



Changing the tempo

We previously set the tempo to T50, but as this will not suit all tunes and tastes why not build in a way of altering the tempo up and down whilst the program is running. We can easily link T+ and T− to the left and right cursor keys so that the left arrow slows the tempo and the right arrow raises the tempo. It is most useful if you can actually hear immediately the result of your action so we continue to PLAY A$ as the tempo changes. If you press any key which is not designated as a note A$ is not updated and

Figure 16.3 Adding a second octave



therefore you will PLAY the old A$ and hence repeat the last note. When you release the cursor key the tempo stays at the current value, so you can use this facility to juggle until you are happy.

```
90 IF A=9 THEN PLAY"T+":GOTO 180
100 IF A=8 THEN PLAY"T-":GOTO 18
0
```

It is possible to crash this routine if you really try hard and make T< 1 or T> 255 but in practice it is unlikely you will reach these limits so we have not bothered to include any limit checks.

Volume control

We could alter the volume in the same way as the tempo, but as the range of values is much smaller (1–31) it is much easier to reach an illegal value. It is therefore better to use an external variable and add this in after STR$ conversion.

```
30 V=15
110 ON ((A=10)-(A=94))+2 GOTO 21
0,120,200
180 PLAY A$+"V"+STR$(V)
200 V=V+1:IF V>31 THEN V=31:GOTO
 180:ELSE 180
210 V=V-1:IF V<1 THEN V=1:GOTO 1
80: ELSE 180
```

The volume is set half way at the start (V= 15) and PLAY "V" + STR$(V) is added in front of A$ in line 180. The keys are sorted by an ON GOTO which looks for the codes for the up and down cursor keys. If neither the up arrow nor the down arrow is pressed then (A= 10)and (A = 94) are both untrue and (0)– (0) = 0 so when 2 is added the result is 2, and the program continues to 120. When the up cursor arrow is pressed (A= 10) is true and (A= 94) is false so (– 1)– (0) + 2 = 1, which goes to 200 which raises the volume unless V> 31. When the down arrow is pressed (A= 10) is false and (A= 94) is true so (0) – (– 1) + 2 = 3 which goes to 210 which lowers the volume unless V< 1.

Changing the octaves

As we have only defined our top octave as being one octave higher than our low octave it is easy to change both octaves at once. The start octave is set at 3 in line 40 and line 120 looks for the codes for keys 1 to 4 (49–52) which change the bottom octave to that number by PLAYing "On".

```
40 O=3
120 ON (A-48) GOTO 220,230,240,2
50
220 PLAY"O1":GOTO 70
230 PLAY"O2":GOTO 70
240 PLAY"O3":GOTO 70
250 PLAY"O4":GOTO 70
```

Envelopes

So far each repeat of each note sounds the same, although most musical instruments actually have a characteristic sound 'envelope'. In simple terms this means that the rate at which the sound rises (attack) and falls (decay) varies, and an approximation of envelope control can be achieved by loops which play each new key with changing sequence of volumes. Several examples are given below and in each case the routine described can be used to replace line 180.

The simplest situation is a steady decay from maximum value.

```
180 FOR E=31 TO 1 STEP-1:PLAY"V"
+STR$(E)+A$:NEXT E
```

Adjust the tempo until you get a reasonable effect and then alter the step size to get more rapid decay.

```
180 FOR E=31 TO 1 STEP-3:PLAY"V"
+STR$(E)+A$:NEXT E
```

A less regular and more interesting effect can be produced by relating the size of the step to the current volume.

```
180 FOR E=31 TO 1 STEP -INT(N/5)
:PLAY"V"+STR$(E)+A$:NEXT E
```

Control of attack can be introduced in the same way by means of a loop which steps up. Attack is normally faster than decay so up steps are bigger than down steps.

```
180 FOR E=1 TO 31 STEP 7:PLAY"V"
+STR$(E)+A$:NEXT E:FOR E=31 TO 1
 STEP -2:PLAY"V"+STR$(E)+A$:NEXT
 E
```

Of course actual values can also be entered if desired to produce any sequence.

```
180 PLAY"V7"+A$+"V14"+A$+"V28"+A
$+"V31"+A$+"V20"+A$+"V10"+A$+"V9
"+A$+"V8"+A$+"V7"+A$+"V6"+A$+"V5
"+A$+"V4"+A$
```

You can also use the < , > , + and – signs in conjunction with V. This rapid size and fall in volume produces rather a tremolo effect.

```
180 PLAY"V7"+A$+"V>"+A$+"V>"+A$+
"V<"+A$+"V<"+A$+"V7"+A$
```

Finally we must point out that it is also possible to change the tempo, note length, or octave to produce interesting sounds by the same methods, although we'll leave you to experiment with those yourselves.

CHAPTER 17
Graphic Music Editor

This graphic music editor gives an excellent demonstration of a combination of the sound and graphics capabilities of the Dragon as it allows you to enter a piece of music, display it in standard musical notation on the screen, and then play it (**Figure 17.1**).

Figure 17.1 Graphic music editor

When entering music we need to consider a number of different factors. A single character on the manuscript tells us more than one thing. The shape of the character tells us the note length and the position on the stave the actual note on the scale and octave. We also need to be able to include sharps and flats. Two modes are provided. In EDIT mode the position is indicated by a flashing cursor which is placed on the line of the stave which corresponds to the current note on the scale. The cursor keys can be used to move this position in any direction. Up and down arrows change the note on the scale, left and right arrows move from your position in the tune, and shifted up and down arrows move you from line to line. The length of note required is chosen by pressing keys 1 to 4. The spacebar is used to delete an unwanted note.

The tune is stored in strings which are sliced to obtain the relevant information for both sound and graphics. Each note is coded by a seven character block.

eg L1203B–, L 402C', or L 802D#

The first three characters define the note length, (eg L12, L 4 or L 8). Note the space when L is less than 10. The next two characters specify the octave, which can be 02 or 03. The sixth character is the note on the scale (A–G) and the last character indicates whether the note is flat (–), natural (') or sharp (#).

If "P" is pressed in EDIT mode then PLAY mode is entered and the tune so far is PLAYed and displayed on the screen. A method of saving your tune is also provided.

Setting up

The first stage of the setting up procedure involves clearing the screen to black on green, clearing 10000 bytes for variables, and setting a number of these. X controls the left/right position on a line, and Y the overall up/down position on the screen, NO is the vertical position on the stave, and LI is the current line of music (1–4)>. Four array elements are set up as PA$(n) to hold the notes entered on each line. Initially these are completely filled by 255 single quote marks (') (CHR$(39). If you try to PLAY a blank space you sometimes get an FC ERROR, but the system is quite happy to PLAY CHR$(39), even though you can't hear it. Filling the string in this way prevents problems when slicing.

```
10 GOTO 690
690 PMODE 4,1:SCREEN1,0:PCLS1:CO
LOR0,1:CLEAR 10000:X=40:Y=48:NO=
7:LI=1:DIM PA$(4):FOR N=1 TO 4:P
A$(N)=STRING$(255,39):NEXT N
```

Graphic parts

We draw all the required graphics parts first and then GET and PUT them (**Figure 17.2**). The picture for each graphic part must be stored in a separate array by GET so a number of arrays are set up.

```
700 DIMSB(0,10):DIMM1(0,10):DIMM
2(0,10):DIMC1(0,10):DIMC2(0,10):
DIMQ1(0,10):DIMQ2(0,10):DIMS1(0,
10):DIMS2(0,10):DIMSP(0,30):DIM
BA(0,10):DIM CU(0,10):DIM SH(0,1
0):DIM FL(0,10)
```

Figure 17.2 Graphic parts

Now the signs for the different note lengths can be drawn. All these have a circle as a basic part so seven are drawn. This completes the drawing of the first one, the semibreve.

```
710 FOR N=20 TO 140 STEP 20:CIRC
LE(N,20),3:NEXT N
```

The other six drawings represent only three actual lengths of note as the position of the tail on these must vary according to their position on the stave. First those with an ascending tail,

```
720 FOR N=40 TO 80 STEP 20:LINE(
N+3,20)-(N+3,10),PSET:NEXT N
```

and then those with a descending tail.

```
730 FOR N=100 TO 140 STEP 20:LIN
E(N-3,20)-(N-3,30),PSET:NEXT N
```

Now we need some black paint to distinguish the quaver and crotchet from the minim,

```
740 PAINT(60,20),0,0:PAINT(80,20
),0,0:PAINT(120,20),0,0:PAINT(14
0,20),0,0
```

and finally we must dash the tail of the quaver.

```
750 LINE(83,10)-(88,13),PSET:LIN
E(137,30)-(142,27),PSET
```

A replacement section of the stave is drawn,

```
760 FOR N=0 TO 16 STEP 4:LINE(15
0,N+15)-(170,N+15),PSET:NEXT N
```

followed by a bar line.

```
770 LINE(180,20)-(180,36),PSET
```

a sharp sign,

```
780 DRAW"BM200,24S4R2U2BM+2,+0;D
2R2BM+0,+2;L2D2BM-2,+0;U2L2"
```

and a flat sign.

```
790 DRAW"BM220,20D10E3H3"
```

We now GET each of these into the appropriate array before passing to the subroutine which draws the stave (this is placed as a subroutine as it is also used by the play routine later).

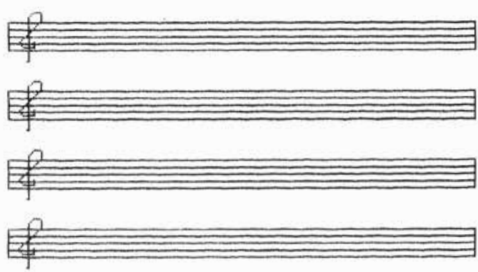
```
800 GET(17,17)-(23,23),SB,G
810 GET(37,10)-(43,23),M1,G
820 GET(57,10)-(63,23),C1,G
830 GET(77,10)-(88,23),Q1,G
840 GET(97,17)-(103,30),M2,G
850 GET(117,17)-(123,30),C2,G
860 GET(137,17)-(148,30),Q2,G
870 GET(150,0)-(170,56),SP,G
880 GET(180,20)-(180,36),BA,G
890 GET(200,22)-(207,28),SH,G
900 GET(220,20)-(224,30),FL,G
910 GOSUB 920:GOTO 60
```

Drawing the stave

The graphics parts are erased and four sets of five lines are constructed down the screen (**Figure 17.3**). The complex treble clef is easily DRAWn after an appropriate Blank Move to set the position.

```
920 PMODE 4,1:SCREEN1,0:PCLS 1:C
LS 1:COLOR 0,1
930 FOR N=40 TO 160 STEP 40
940 FOR M=0 TO 16 STEP 4
950 LINE(0,N+M)-(255,N+M),PSET
960 NEXT M
970 LINE(0,N)-(255,N+16),PSET,B
980 DRAW"BM10,"+STR$(N+22)+"RU25
E3R3FD4G12DF2R6U3"
990 NEXT N
1000 RETURN
```

Figure 17.3 The stave



On RETURN we jump back to the program proper in line 60.

Cursor and keycheck

INKEY$ is read into A$ and then we GET a square of the screen around coordinates X,Y into CU and immediately PUT it back with PRESET. This inverts the screen display in that area. After a short delay CU is PUT back with PSET to recreate the original display. If no key is pressed this flashing cursor sequence is repeated. If a key is pressed a check is made to see if the current position is too far to the left (X< 40) or right (X> 240).

```
60 A$=INKEY$:GET(X-5,Y-5)-(X+5,Y
+5),CU,G:PUT(X-5,Y-5)-(X+5,Y+5),
CU,PRESET:FOR N=1 TO 50:NEXT N:P
UT(X-5,Y-5)-(X+5,Y+5),CU,PSET:IF
 A$="" THEN 60 ELSE IF X<40 OR X
>240 THEN GOTO 80
```

Note lengths

If the position is valid then the VALue of the key pressed is taken. Only number keys have a VALue so this separates the number keys from other keys. Keys 1 to 4 are used to indicate note lengths from semibreve to quaver and only these will branch in the ON GOTO to the lines which draw the characters.

```
70 A=VAL(A$):ON A GOTO 210,220,2
30,240
```

The semibreve is easily dealt with as it looks the same no matter where it appears on the stave. Note that the array is PUT. . .,AND rather than OR to produce superimposition as the screen display is inverted.

```
210 PUT(X-3,Y-3)-(X+3,Y+3),SB,AN
D:GOTO 250
```

For the other note lengths the current note position on the scale must be checked to determine if the tail should go up or down. If you have not changed the cursor position then the note position (NO) will still be 7.

```
220 IF NO<7 THEN PUT(X-3,Y-10)-(
X+3,Y+3),M1,AND:GOTO 250:ELSE PU
T(X-3,Y-3)-(X+3,Y+10),M2,AND:GOT
O 250
230 IF NO<7 THEN PUT(X-3,Y-10)-(
X+3,Y+3),C1,AND:GOTO 250:ELSE PU
T(X-3,Y-3)-(X+3,Y+10),C2,AND:GOT
O 250
240 IF NO<7 THEN PUT(X-3,Y-10)-(
X+8,Y+3),Q1,AND:GOTO 250:ELSE PU
T(X-3,Y-3)-(X+8,Y+10),Q2,AND:GOT
O·250
```

Adding to the strings

Once the screen display has been updated an ON GOSUB related to the note (NO) on the scale sets NO$ to the correct octave and note format for PLAYing. On RETURN PL$ is built up by adding "L" to four times the VALue of the key pressed (A*4) and NO$.

```
250 ON NO GOSUB260,270,280,290,3
00,310,320,330,340,350,360,370,3
80,390:PL$="L"+RIGHT$(STR$(A*4),
2)+NO$:GOSUB 400:X=X+20::GOTO 20
260 NO$="O2C;":RETURN
270 NO$="O2D;":RETURN
280 NO$="O2E;":RETURN
290 NO$="O2F;":RETURN
300 NO$="O2G;":RETURN
310 NO$="O2A;":RETURN
320 NO$="O2B;":RETURN
330 NO$="O3C;":RETURN
340 NO$="O3D;":RETURN
350 NO$="O3E;":RETURN
```

```
360 NO$="O3F'":RETURN
370 NO$="O3G'":RETURN
380 NO$="O3A'":RETURN
390 NO$="O3B'":RETURN
```

The subroutine at 400 is now called. This inserts the current string (PL$) into the total string (PA$(LI)). XS is calculated from the current screen position and defines the breakpoint between two notes. I and SF are used if PL$ is a sharp or flat (see later).

```
400 XS=(((X-20)/20)*7)+I:PA$(LI)
=LEFT$(PA$(LI),XS-SF)+PL$+MID$(P
A$(LI),XS+8,LEN(PA$(LI))-4):RETU
RN
```

Finally the screen position is updated (X = X + 20) and the program loops back to line 20.

Limit tests

After each key press checks are made to ensure that the new cursor position is within limits, and XA (distance of current move) is reset to zero.

```
20 IF X+XA<40 THEN X=X-XA ELSE X
=X+XA
30 IF X+XA>250 THEN X=X-XA ELSE
X=X+XA
40 IF X>240 THEN X=240
50 XA=0
```

Other keys

If a key which is not a number in the range 1 to 4 is pressed then a series of other routines may be called.

Cursor keys

Logic tests convert left/right cursor key movement into increases in XA (X axis position), and up/down cursor key movement into changes in NO (note position on current line).

```
80 A=ASC(A$):XA=(10*((A=8)-(A=9)
)):NO=NO+(((A=10)-(A=94)))
```

If the note position falls outside limits it is reset to the limit and then the overall Y coordinate is calculated from the current line (LI) and note (NO).

```
90 IF NO<1 THEN NO=NO+1
100 IF NO>14 THEN NO=NO-1
110 Y=(LI*40)+22-(NO*2)
```

"B" = bar line

If "B" is pressed a bar line is inserted. This is purely decorative and is not added to the string.

```
120 IF A$="B" THEN PUT((X-15,(LI*
40))-(X-15,(LI*40)+16),BA,PSET
```

Shifted cursor

Shifted up and down cursor keys produce a movement from line to line, provided the limits are not exceeded. The start position is reset to the left hand end, and the overall Y coordinate updated.

```
130 IF A=91 AND LI<4 THEN LI=LI+
1:X=40:Y=(LI*40)+22-(NO*2)
140 IF A=95 AND LI>1 THEN LI=LI-
1:X=40:Y=(LI*40)+22-(NO*2)
```

<spacebar> = delete

Pressing the space bar PUTS the spare section of stave with PSET over the note to be deleted, thus removing it from the screen. At the same time the old note is deleted from PA$(LI) by replacing it with a series of CHR$(39).

```
150 IF A=32 THEN PUT(X-10,(LI*40
)-15)-(X+10,(LI*40)+31),SP,PSET:
PL$="'''''''":GOSUB 400
```

"#" = sharp

The hash sign is used to indicate a sharp and this is PUT . . .,PSET rather than AND to make it clearer. The hash sign appears to the left of the current cursor position and as SF is set to 1 and I to 7 the hash sign is added to the note to the left of the current cursor position, replacing the trailing CHR$(39) in the seven unit block.

```
160 IF A$="#" THEN PUT(X-11,Y-3)
-(X-4,Y+3),SH,PSET:PL$="#":SF=1:
I=7:GOSUB 400:SF=0:I=0
```

"–" = flat

The minus sign indicates a flat and operates in the same way.

```
170 IF A$="-" THEN PUT(X-7,Y-7)-
(X-3,Y+3),FL,AND:PL$="-":SF=1:I=
7:GOSUB 400:SF=0:I=0
```

"P" = play

"P" leads to the PLAY routine, which first calls the subroutine at 920 which draws the blank manuscript.

```
180 IF A$="P" THEN GOSUB 410
410 GOSUB 920
```

Each line is considered in turn, with the start position (X2) being first set to coordinate 40.

```
420 FOR PL=1 TO 4:X2=40
```

The string is sliced from position 6 (seventh character) to the end in blocks of seven, and each block is PLAYed.

```
430 FOR X1=6 TO 255 STEP 7
440 PLAYMID$(PA$(PL),X1,7)
```

The end of the actual notes on a line is detected by the presence of two consecutive blocks of CHR$(39).

```
450 IF MID$(PA$(PL),X1,7)="'''''
''" THEN FL=FL+1 ELSE FL=0
460 IF FL>2 THEN NEXT PL:RETURN
```

To recreate the graphics the string segment must be decoded. First we must extract the last but one character as NO$.

```
470 NO$=MID$(PA$(PL),X1+7,1)
```

NO$ is compared against the scale of notes in VN$ with INSTR to set N1 to the number of the note on the scale. The actual Y1 position can now be calculated.

```
480 VN$="CDEFGAB":N1=INSTR(1,VN$
,NO$):Y1=(PL*40)+22-(N1*2)
```

Octave can only be 2 or 3 so we only need a check for 3 in position five to know whether to move Y1 up for the higher octave.

```
490 IF MID$(PA$(PL),X1+6,1)="3"
THEN Y1=Y1-14
```

The length of the note is extracted as the second and third characters (LN$) and this is converted to a number by taking the VALue.

```
500 LN$=MID$(PA$(PL),X1+3,2)
510 LN=VAL(LN$)
```

Now we divide the actual note length by 4 to GOTO the routines to actually PUT the notes. These are very similar to those described before.

```
520 ON (LN/4) GOTO 540,550,560,5
70
530 GOTO 580
540 PUT(X2-3,Y1-3)-(X2+3,Y1+3),S
B,AND:GOTO 580
550 IF N1<7 THEN PUT(X2-3,Y1-10)
-(X2+3,Y1+3),M1,AND:GOTO 580:ELS
E PUT(X2-3,Y1-3)-(X2+3,Y1+10),M2
,AND:GOTO 580
560 IF N1<7 THEN PUT(X2-3,Y1-10)
-(X2+3,Y1+3),C1,AND:GOTO 580:ELS
E PUT(X2-3,Y1-3)-(X2+3,Y1+10),C2
,AND:GOTO 580
570 IF N1<7 THEN PUT(X2-3,Y1-10)
-(X2+8,Y1+3),Q1,AND:GOTO 580:ELS
E PUT(X2-3,Y1-3)-(X2+8,Y1+10),Q2
,AND:GOTO 580
```

If the last character is "#" or "–" then the sign is PUT in the appropriate position.

```
580 IF MID$(PA$(PL),X1+8,1)="#"
THEN PUT(X2-11,Y1-3)-(X2-4,Y1+3)
,SH,PSET
590 IF MID$(PA$(PL),X1+8,1)="-"
THEN PUT(X2-7,Y1-7)-(X2-3,Y1+3),
FL,AND
```

The left/right coordinate (X2) is incremented by 20 and the next note taken.

```
600 X2=X2+20:NEXT X1,PL:RETURN
```

"S" = save/load

"S" leads to a save/load routine which allows you to SAVE the strings on tape as ASCII files and reLOAD them to recreate both sound and graphics. After SAVEing the cursor is returned to the top of the hi-res screen.

```
190 IF A$="S" THEN 610
610 CLS:PRINT@228,"";:INPUT"DO Y
OU WISH TO LOAD OR SAVE";Z$
620 IF LEFT$(Z$,1)="L" THEN 660
ELSE IF LEFT$(Z$,1)<>"S" THEN SC
REEN1,0:GOTO 20
630 INPUT"FILE NAME";NA$:OPEN "O
",#-1,NA$
640 FOR LI=1 TO 4:PRINT#-1,PA$(L
I);:NEXT LI:CLOSE#-1
650 LI=1:Y=48:X=40:NO=7:GOTO 20
```

After LOADing the cursor position is set to the top and the PLAY routine automatically called.

```
660 INPUT"FILE NAME";NA$:OPEN"I"
,#-1,NA$
670 FOR LI=1 TO 4:INPUT#-1,PA$(L
I);:NEXT LI:CLOSE#-1
680 LI=1:Y=48:X=40:NO=7:A$="P":G
OTO 180
```

Any other key will fall through to line 200 and return to 20.

```
200 GOTO 20
```

CHAPTER 18
Beyond BASIC

Although you can do a great many things with BASIC programs there will always remain certain things that you would like to do but which you find impossible. Often you feel that you cannot move things fast enough, or that your programs take up too much memory. The solution to these problems lies deeper inside your Dragon where you must get to grips directly with its internal workings. There is not enough room to go into detail here on this complex subject but we will give you some examples and pointers to show what is possible, whet your appetite, and perhaps start you on your way to exploring this fascinating area.

Before we start you must clearly understand how your Dragon carries out your commands. First you must realise that the 6809 microprocessor at its heart can only really understand instructions if these are given to it as numbers, and that the BASIC interpreter converts programs written in pseudo-English into these numbers so that they can be acted upon. As BASIC is an interpreted language your lines are re-read every time the program passes through them. If you can alter these lines whilst the program is running then you will be able to change the program itself.

You have already met the commands PEEK and POKE in other contexts, but now let us look at another way these can be very useful. In case you have forgotten let us remind you that PEEK tells you what is in a particular memory location and POKE will put any number from 0 to 255 into a memory location.

Poking into your program

You will remember that when we developed the program to write text on the hi-res screen using GET and PUT we had to type in separate DIM, GET and PUT lines for each character as the BASIC interpreter will not allow you to rename the array you want to use. That is very tedious, of course, but in addition all those lines eat up memory. There must be some solution to this problem so why not do a little PEEKing around in the memory locations which contain your BASIC program so you can see how this is stored. We will position the key lines DIM, GET and PUT at the end of the program and add RETURN to each so that we can call them as subroutines.

```
100 DIM CZ(10):RETURN
110 GET(G1,E1)-(G2,E2),CZ,G:RETU
RN
120 PUT(P1,U1)-(P2,U2),CZ,NOT:RE
TURN
```

Finding the end

As these are the last lines in the program we know that they must be just before the end of the program, and if we look in locations 27 and 28 we can find out where that is, as these two bytes always contain the address of the end of the current program. We will define this position as the variable EN.

```
10 EN=PEEK(27)*256+PEEK(28)
```

To look at our program we must PEEK at the locations before this marker. The following line will look at the last 69 memory locations of the program and will print out memory location address (N), negative offset from the end marker (EN-N), number in that location (PEEK(N)), and the character corresponding to that number (CHR$(PEEK(N)).

```
20 FOR N=(EN-68)TO EN:PRINT N;"
   ";(EN-N);"      ";PEEK(N);"
   ";CHR$(PEEK(N)):NEXT N
```

If you RUN this now you will see four changing columns which are something like those shown in **Table 18.1**. (In addition to the characters shown on the print-out in our table some graphics characters will also appear at the right of the screen. These are not acceptable to our printer and it has therefore ignored them). If you find that the actual location numbers differ from those printed then you have probably got a different number of spaces in your program, or you have previously used PCLEAR to set a different number of graphics pages from the default of 4. This does not really matter for the moment. Look closely at the last column of our table, LIST the program on the screen, and compare the characters with the last three lines of the program. You should be able to recognise parts of it. For example locations 7842 and 7843 contain C and Z, and 7838 and 7839 contain U and 2. Much of the rest of the output appears as garbage because the system uses certain numbers to define BASIC commands instead of storing each character of the command word. The key points as far as we are concerned are those that name the array to be used. If you look at all the locations listed you will find that these appear as follows:

7787 C
7788 Z (in the DIM)

7817 C
7818 Z (in the PUT)

7842 C
7843 Z (in the GET)

Table 18.1

PEEKING AT THE END OF PROGRAMS

address	offset	PEEK	CHR$(PEEK)
7819	33	145	
7820	32	0	
7821	31	30	
7822	30	170	[illegible]
7823	29	0	
7824	28	120	x
7825	27	180	Ü
7826	26	40	(
7827	25	80	P
7828	24	49	1
7829	23	44	,
7830	22	85	U
7831	21	49	1
7832	20	41	)
7833	19	196	T
7834	18	40	(
7835	17	80	P
7836	16	50	2
7837	15	44	,
7838	14	85	U
7839	13	50	2
7840	12	41	)
7841	11	44	,
7842	10	67	C
7843	9	90	Z
7844	8	3	
7845	7	44	,
7846	6	192	P
7847	5	58	:
7848	4	145	
7849	3	0	
7850	2	0	
7851	1	0	

```
7852     0     69    E
7787    68     67    C
7788    67     90    Z
7789    66     40    (
7790    65     49    1
7791    64     48    0
7792    63     41    )
7793    62     58    :
7794    61    145
7795    60      0
7796    59     30
7797    58    144
7798    57      0
7799    56    110    n
7800    55    179    ö
7801    54     40    (
7802    53     71    G
7803    52     49    1
7804    51     44    ,
7805    50     69    E
7806    49     49    1
7807    48     41    )
7808    47    196    T
7809    46     40    (
7810    45     71    G
7811    44     50    2
7812    43     44    ,
7813    42     69    E
7814    41     50    2
7815    40     41    )
7816    39     44    ,
7817    38     67    C
7818    37     90    Z
7819    36     44    ,
7820    35     71    G
7821    34     58    :
```

Changing the array names

Now try POKING location 7843 with different numbers (as direct commands) and then LISTing line 120

```
eg    POKE 7843,65
```

```
120 PUT(P1,U1)-(P2,U2),CA,NOT:RE
TURN
```

Look closely and you will see that the array name used in the PUT command is now CA instead of CZ! (65 is the ASCII code for A).

```
POKE 7843,66
```

will produce:

```
120 PUT(P1,U1)-(P2,U2),CB,NOT:RE
TURN
```

You can also POKE the same place by defining it in terms of a negative offset from the end of program marker. The offset for 7843 is −9.

```
POKE EN-9,67
```

gives

```
120 PUT(P1,U1)-(P2,U2),CC,NOT:RE
TURN
```

The offsets for Z in the DIM and GET lines are −67 and −37, respectively.

The absolute value of the end of program marker will change if you alter the length of the program but if you define the position you want to POKE as an offset from this then you will always produce the correct result *provided that you do not alter the program after your key point*. This is the reason we put these lines at the end of the program. No matter how much you add before them the offset from the end will always be correct. Even if you RENUM to change the program line numbers it will work. Another very important factor is that the POKE can also be made from within the program. Add this line, RUN and LIST.

```
30 POKE EN-9,68
```

gives

```
120 PUT(P1,U1)-(P2,U2),CD,NOT:RE
TURN
```

So now we can reach the parts other methods cannot reach and modify our program as it is actually running how can we apply this in practice?

What can I POKE

POKEing around in your program is not a particularly dangerous occupation. The worst that can happen is that the system crashes and you lose your program, but perhaps you'd better CSAVE a copy now if you like to POKE at random. The most straightforward way to apply this POKE idea to dealing with GETting and PUTting characters would seem to be to use the actual character in the array name. Unfortunately life is not quite that simple as many of the characters have other meanings when in a program line. For example the colon (:) is used to separate commands.

Think back to what is normally a valid name for a variable and you will know which characters you can and cannot use. One or two characters are allowed, the first one must be an upper case (capital) letter, and the second can be a letter or a number. If you use only one character to define the array then you are limited to 26 different arrays. On the other hand with a letter followed by a number you can have 26*10 = 260 arrays different arrays, and with two letters another 26*26 = 676 which gives a total so large that there should be no difficulty in making enough arrays for all the characters you might need (even if you do nothing but design characters for the rest of your life!)

Rather than getting into discussions on the best way to code each character we will just give you the listing below which contains a straightforward modification which changes the array name for the characters A to Z, as an example, and we leave the rest to you. The reduction in the memory requirement is dramatic as the whole program can now be fitted in well under 1K. Don't forget that you can also POKE the 'action' of the PUT command in the same way to give global changes in the way the array affects the screen.

```
1 EN=PEEK(27)*256+PEEK(28)
10 PMODE 4,1:SCREEN1,0:PCLS1:COL
OR0,1:CLOADM"MCHAR2"
15 FOR N=48 TO 57:POKE EN-9,N:GO
SUB 2000:NEXT N
16 FOR N=65 TO 90:POKE EN-9,N:GO
SUB 2000:NEXT N:DIM BL(1)
40 X=1:S=7:A=5:B=7
50 FOR N=48 TO 57:POKE EN-22,N:G
OSUB 1900:X=X+S8:NEXT N
55 FOR N=65 TO 90:POKE EN-22,N:G
OSUB 1900:X=X+S:NEXT N
500 PCLS:X=2:Y=0:S=6:T=8:XS=2:XE
=253:YS=0:YE=191
510 C$=INKEY$:PUT(X,Y)-(X+A,Y+B)
,BL,NOT:PUT(X,Y)-(X+A,Y+B),BL,NO
```

```
T:IF C$="" THEN 510 ELSE IF C$="
@" THEN 1620 ELSE C=ASC(C$):IF C
<48 OR C>57 THEN IF C<65 OR C>91
 THEN 1500
520 POKE EN-50,C:GOSUB 1800
530 IF FL=1 THEN PUT(X,Y)-(X+A,Y
+B+1),BL,NOT
540 X=X+S:IF X>XE THEN X=1:Y=Y+T
:IF Y>YE THEN 1610
550 GOTO 510
1500 IF C=32 THEN PUT(X,Y)-(X+A,
Y+B),BL,PRESET:X=X+Q:GOTO 510
1510 IF C=8 THEN X=X-S:GOTO 1560
1520 IF C=9 THEN X=X+S:GOTO 1560
1530 IF C=94 THEN Y=Y-T:GOTO 156
0
1540 IF C=10 THEN Y=Y+T:GOTO 156
0
1550 GOTO 510
1560 IF X<XS THEN X=XS
1570 IF X>XE THEN X=XS:Y=Y+T
1580 IF Y<YS THEN Y=YS
1590 IF Y>YE THEN Y=Y-T
1600 GOTO 510
1620 IF FL=1 THEN FL=0:GOTO 510
ELSE FL=1:GOTO510
1800 PUT(X,Y)-(X+A,Y+B),CT,PSET:
RETURN
1900 GET(X,Y)-(X+A,Y+B),CZ,G:RET
URN
2000 DIM CZ(1):RETURN
```

Hidden graphics modes

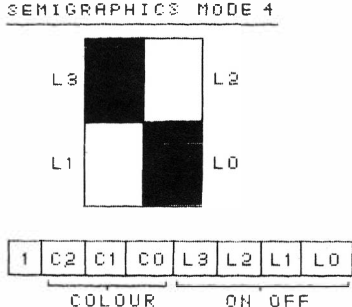
The Video Display Generator chip used in the Dragon is a general purpose device which is also capable of producing extra display modes which are not available through Microsoft Colour BASIC. However to be able to work in these modes you must take direct control of the VDG, which means life gets complicated and you must learn how the various modes operate.

Semigraphics

The low-resolution graphics display available through BASIC is actually semigraphics mode 4, in which each character position of the text display is divided into 4 elements (**Figure 18.1**), and each character position takes up

one byte. There are three parts to this byte. The first bit is 1, which indicates that this is a semigraphics mode, the next three bits code for the colour, and the last four bits indicate whether each element is on or off. Since the colour code can only be set for the whole character position you cannot set different elements to different colours.

Figure 18.1



This mode is set up automatically by the BASIC interpreter when you are using the text screen but the other semigraphics modes can only be obtained by POKEing to certain addresses. Even when these different modes have been set up in this way you can only alter the screen display by POKEing to screen memory (or using machine code) and in practice this means that these modes are rather fiddly to use. Details of the necessary POKEs are given in the examples below.

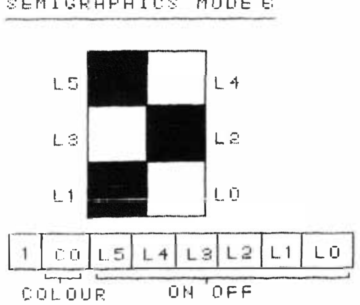
Semigraphics mode 6

Semigraphics mode 6 divides each character position into six elements and uses the same amount of memory as semigraphics mode 4 (**Figure 18.2**). The first bit is set to indicate semigraphics, and only the second bit is used to code for colour, so that only two colours can be indicated. The rest of the bits indicate on/off status and only one colour can be used in each character position. The colour set is either blue/red (if bit 4 of 65314 is set) or magenta/orange (if bit 4 of 65314 is 0). This mode is not particularly useful

but the following demonstration shows it filling the screen with each of the graphics characters in turn.

```
10 POKE 65314,PEEK(65314)+16
20 POKE 65472,0:POKE 65474,0:POK
E 65476,0
30 GOSUB 100
40 POKE 65314,PEEK(65314)+24
50 GOSUB 100
60 GOTO 60
100 FOR CH=128 TO 255
110 FOR SC=1024 TO 1535
120 POKE SC,CH
130 NEXT SC,CH
140 RETURN
```

Figure 18.2



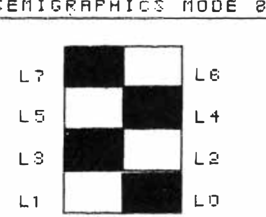
Semigraphics modes 8, 12 and 24

Semigraphics modes 8, 12 and 24 are more interesting. They give you control of smaller elements (64*64, 64*96, and 64*192 pixels, respectively) and allow you to use all eight colours in the same character position. They also include normal text in the display (something which of course you cannot normally do in the BASIC hi-res modes). In each case the character position is divided into 'N/2' rows of two elements **(Figures 18.3–18.5)** and

any of the eight colours can be specified for each row. However as one byte is used to code for each row more memory is needed (2–6K). In all these bytes the first bit is set to 1, the next three used to code for colour, and the last four to indicate on/off status. The following demonstration sets pixels to random colours in each mode to show the degrees of resolution available.

```
10 CLS8:PMODE 4,1:PCLS 1
20 POKE 65472,0:POKE 65475,1:POK
E 65476,0:H=63:GOSUB 100
30 POKE 65472,0:POKE 65474,0:POK
E 65477,1:H=95:GOSUB 100
40 POKE 65472,0:POKE 65475,1:POK
E 65477,1:H=191:GOSUB 100
50 GOTO 50
100 FOR LN=1 TO H
110 FOR CL=1 TO 31
120 POKE 1024+LN*32+CL,RND(127)+
128
130 NEXT CL,LN
300 CLS8:PCLS 1:RETURN
```

Figure 18.3

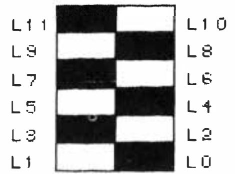


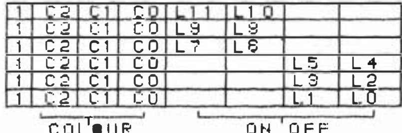
1	C2	C1	C0	L7	L6		
1	C2	C1	C0	L5	L4		
1	C2	C1	C0			L3	L2
1	C2	C1	C0			L1	L0

COLOUR ON OFF

Figure 18.4

SEMIGRAPHICS MODE 12





1	C2	C1	C0	L11	L10		
1	C2	C1	C0	L9	L8		
1	C2	C1	C0	L7	L6		
1	C2	C1	C0			L5	L4
1	C2	C1	C0			L3	L2
1	C2	C1	C0			L1	L0

CLS8, PMODE 4,1 and PCLS1 are included as a convenient way of wiping the text screen and the first four pages of video memory clear before starting, and between modes, as there is no built-in screen-clearing routine for these non-implemented modes. If you use these modes you can have eight colours in 'hi-resolution' but you will need to do a lot of planning to get your display correct without all those useful graphics commands available in Microsoft Color BASIC.

Including text

The normal text is generated within the VDG when a number less than 128 is received but to put this text on the screen in these special modes you must send the character code repeatedly. The number of repeats is the same as the number of the semigraphics mode, with a horizontal slice of the character being sent at each repeat. If you add these modifications each mode will be labelled. The numbers are inverted as the screen codes are not the same as the ASCII codes (as explained earlier).

Figure 18.5

SEMIGRAPHICS MODE 24

L23 L21 L19 L17 L15 L13 L11 L9 L7 L5 L3 L1

L22 L20 L18 L16 L14 L12 L10 L8 L6 L4 L2 L0

1	C2	C1	C0	L23	L22		
1	C2	C1	C0	L21	L20		
1	C2	C1	C0	L19	L18		
1	C2	C1	C0	L17	L16		
1	C2	C1	C0	L15	L14		
1	C2	C1	C0	L13	L12		
1	C2	C1	C0			L11	L10
1	C2	C1	C0			L9	L8
1	C2	C1	C0			L7	L6
1	C2	C1	C0			L5	L4
1	C2	C1	C0			L3	L2
1	C2	C1	C0			L1	L0

COLOUR ON OFF

```
10 CLS8:PMODE 4,1:PCLS 1:M=255:S
T=1024
20 POKE 65472,0:POKE 65475,1:POK
E 65476,0:H=63:R=8:M$="SEMIGRAPH
ICS 8":GOSUB 100
30 POKE 65472,0:POKE 65474,0:POK
E 65477,1:H=95:R=12:M$="SEMIGRAP
HICS 12":GOSUB 100
40 POKE 65472,0:POKE 65475,1:POK
E 65477,1:H=191:R=24:M$="SEMIGRA
PHICS 24":GOSUB 100
200 CO=0:FOR N=1 TO LEN(M$):M=AS
C(MID$(M$,N,1))
210 FOR CH=1 TO R*16 STEP 32
```

```
220 POKE ST+CH+CO,M
230 NEXT CH:CO=CO+1:NEXT N
240 SOUND 1,16
```

True graphics

Three extra true graphics modes can also be obtained with suitable POKEs.

64 × 64	four colour mode (1K video memory)
64 × 128	two colour mode (1K video memory)
64 × 128	four colour mode (2K video memory)

These are all of lower resolution than the PMODEs available through BASIC and as they are really of interest only we have omitted details on setting them up.

Calling machine code subroutines

Machine code is the ultimate language of the microprocessor and even if you do not go to the extreme of writing programs entirely in machine code you can use subroutines written in it to improve your programs. We cannot even attempt to go into the details of 6809 code here, as that would take at least one whole book on its own (if you think that BASIC is complicated then you will soon realise that machine code is rather like ancient Sumarian hieroglyphics by comparison). We will therefore just explain how you can use machine code subroutines in your BASIC programs, and give a few examples of sound and graphics routines. All data is given in hexadecimal base. That may make things look even more complicated but if you want to get into machine code you are going to have to get used to it sometime, so you might as well start now!

CLEARing space

First you need to reserve room in memory to store the machine code you write so that it cannot be obliterated by BASIC programs or variables. This is done with the CLEAR command, which is also used to reserve string space for BASIC programs. To reserve space for machine code a second parameter must be added, which limits the highest address that BASIC can use.

Thus:

```
10 CLEAR 200
```

reserves 200 bytes for strings and:

```
10 CLEAR 200,&H6000
```

reserves 200 bytes for strings and the area above address &H6000 for machine code routines.

Entering machine code routines

If you are going to do much work with machine code then you should invest in an editor/assembler but in the meantime this little program will allow you to enter code quite painlessly. There is no need to type '&H' to indicate hexadecimal numbers as this is added automatically. (Assembler listings are also included for the fortunate).

```
1000 CLS:PRINT"START ADDRESS";:I
NPUT ST$:ST=VAL("&H"+ST$)
1010 PRINT"START ENTERING DATA"
1020 PRINT HEX$(ST);:INPUT A$
1030 POKE ST,VAL("&H"+A$)
1040 ST=ST+1
1050 GOTO 1020
```

One major difficulty with machine code is that there are no error-trapping routines built in, so if you make a mistake entering the data the whole thing can quite easily crash.

Simple sounds

Sound is turned on by loading a byte to address &HFF23, and the tone sounded depends on the value loaded into &HFF20. The duration depends on a time delay which you build into the program. This simple routine just makes a single sound. When you have entered the numbers in the second column of **Table 18.2** from address &H6000 with the loader program above you call it by EXECuting from the start address. You should be able to define up to 10 separate machine code routines on the Dragon within the USR n function but, due to a bug in the ROM, USR 0 is always called no matter what number you specify. Where no parameters need to be passed to the routine this causes no problem, as you can simply EXECute the starting address of the routine to call it.

```
100 EXEC&H6000
```

If you RUN this BASIC program it will make a single sound and then report back with OK. If you add 30 GOTO 20 it will repeat until you press BREAK. Where you need to be able to pass parameters to a machine code routine the simplest thing is to EXECute it after POKEing values into it. The tone value used is stored at address &H6009, and the duration as a two byte number at addresses &H6006 and &H6007, so try experimenting by POKEing in different values.

eg

```
20 POKE &H6007,&HAF
```

Table 18.2

```
DISASSEMBLE FROM=6000 TO=6015
6000 86 3F                LDA   #3F
6002 B7 FF 23             STA   $FF23
6005 8E 00 FF             LDX   #00FF
6008 C6 5F                LDB   #5F
600A F7 FF 20             STB   $FF20
600D 5C                   INCB
600E 26 FA                BNE   600A
6010 30 1F                LEAX  -1,X
6012 26 F4                BNE   6008
6014 39                   RTS
```

If you are too lazy to think of values then try

```
20 POKE &H6007,RND(&HFF)
```

although we warn you that it will sound a bit like morse code!
If you add

```
30 POKE &H6009,RND(&HFF)
```

it will sound a little more like the orchestra tuning up.

Saving your routines

The area of memory reserved for machine code is not saved by a normal BASIC program CSAVE so you must use CSAVEM and take into account the address and length of the program. For example this first routine can be saved by:

```
CSAVEM"sound",&H6000,&H6014,&H14
```

Sound effects

Machine code allows you to make more interesting sounds as these can change tone very rapidly. For example the listing in **Table 18.3** produces a 'phaser' type sound. It is entered from &H6100. The BASIC routine below calls it whenever a key is pressed, but POKEs different values into it according to whether A or B is pressed to produce two different sounds.

```
20 IF PEEK(337)=255 THEN 20 ELSE
 I=PEEK(135)
30 IF I=65 THEN POKE&H6001,FF EL
SE IF I=66 THEN POKE&H6001,3F EL
SE 20
40 EXEC&H6100
50 GOTO 20
```

Table 18.3

```
DISASSEMBLE FROM=6100 TO=6113
6100 86 3F          LDA   #3F
6102 B7 FF 23       STA   $FF23
6105 1F 89          TFR   A,B
6107 F7 FF 20       STB   $FF20
610A 5C             INCB
610B 26 FA          BNE   6107
610D 4C             INCA
610E 2A 01          BPL   6111
6110 4F             CLRA
6111 20 F2          BRA   6105
```

Sound tables

It is often useful to be able to set up a sequence of tones to be played, and these are best organised in a 'sound table' in memory. The program in **Table 18.4** starts from &H6200 and reads tone bytes from **Table 18.5** which starts at &H6250 and continues to sound these in sequence until it finds a zero. Use the loader program to enter some values into this table and listen to the effect (you will have all the space up to &H64FF available). To speed things up POKE a smaller value into &H620B.

Inverting the text screen

Normal and inverted characters on the text screen can easily be interconverted with the listing in **Table 18.6** which makes an EOR (exclusive OR) of each character on the text screen with &H40. The BASIC program below will invert the screen every time a key is pressed thus alternating between the two forms.

```
20 I$=INKEY$:IF I$="" THEN 20
30 EXEC&H6500
40 GOTO 20
```

Table 18.4

```
DISASSEMBLE FROM=6200 TO=6223
6200 86 3F              LDA   #3F
6202 B7 FF 23           STA   $FF23
6205 10 8E 62 50        LDY   #6250
6209 8E 00 80           LDX   #0080
620C E6 A0              LDB   ,Y+
620E C1 00              CMPB  #00
6210 27 13              BEQ   6225
6212 1F 98              TFR   B,A
6214 F7 FF 20           STB   $FF20
6217 5C                 INCB
6218 26 FA              BNE   6214
621A 1F 89              TFR   A,B
621C 30 1F              LEAX  -1,X
621E 26 F4              BNE   6214
6220 20 E7              BRA   6209
6222 39                 RTS
```

Table 18.5

```
SOUND TABLE

 &H
 --

6250     A3
6251     32
6252     A3
6253     37
6254     84
6255     56
6256     25
6257     89
6258     FF
6259     B5
6260     00
```

No doubt you will be impressed by the speed of this routine which is virtually instantaneous. If you want to invert only part of the screen change the two byte start and end address values in &H6501/&H6502 and &H650A/&H650B, respectively. For example if you POKE &H650A with &H05 then only the top half of the screen will invert.

Table 18.6

```
DISASSEMBLE FROM=6500 TO=650F
6500 8E 04 00         LDX    #0400
6503 A6 84            LDA    ,X
6505 88 40            EORA   #40
6507 A7 80            STA    ,X+
6509 8C 06 00         CMPX   #0600
650C 25 F5            BCS    6503
650E 39               RTS
```

Partial PCLS

The routine in **Table 18.7** allows you to fill certain bytes of the hi-res graphics screens with any number. The main use is in clearing parts of the screen or setting up a particular pattern. The routine places the values in &H6601 and &H6603 into consecutive bytes of the screen. This is particularly fast as it is done in one movement by treating the 8 bit A and B registers as a single 16 bit D register. The start address of the area to be filled is at &H6605/&H6606 and the end address at &H660A/&H660B.

```
20 PMODE 3,1:SCREEN 1,0
30 EXEC&H6600
40 GOTO 40
```

Table 18.7

```
DISASSEMBLE FROM=6600 TO=660F
6600 86 00            LDA    #00
6602 C6 55            LDB    #55
6604 8E 06 00         LDX    #0600
6607 ED 81            STD    ,X++
6609 8C 17 FF         CMPX   #17FF
660C 25 F9            BCS    6607
660E 39               RTS
```

If zeros are POKEd into &H6601 and &H6603 the top three quarters of the screen will be cleared as for PCLS 1, and if &HFF is POKEd the effect will be as PCLS 4. If &H6601 is POKEd with zero and &H6603 with &HFF the result is red and green stripes. Experiment with other values remembering that each screen point is controlled by a pair of bits in PMODE 3.

Scrolling

Although the text screen scrolls upwards automatically when the PRINT position reaches the bottom no scrolling of the hi-res screen is provided in Color Basic. The listing in **Table 18.8** provides upward scrolling of the screen, and the routine in **Table 18.9** provides a similar downward effect. The overall effect depends on the values POKEd into tables stored at &H6740 and &H6840, respectively. These values can be stored as DATA and POKEd in when required. The example below gives smooth control over the up and down motion of a floating circle with the up and down cursor keys.

Table 18.8

```
DISASSEMBLE FROM=6700 TO=6716
6700 FC 67 40          LDD   $6740
6703 10 BE 67 42       LDY   $6742
6707 FE 67 44          LDU   $6744
670A 4C                INCA
670B AE A1             LDX   ,Y++
670D AF C1             STX   ,U++
670F 5A                DECB
6710 26 F9             BNE   670B
6712 4A                DECA
6713 26 F6             BNE   670B
6715 39                RTS
```

Table 18.9

```
DISASSEMBLE FROM=6750 TO=6766
6750 FC 67 90          LDD   $6790
6753 10 BE 67 92       LDY   $6792
6757 FE 67 94          LDU   $6794
675A 4C                INCA
675B AE A3             LDX   ,--Y
675D AF C3             STX   ,--U
675F 5A                DECB
6760 26 F9             BNE   675B
6762 4A                DECA
6763 26 F6             BNE   675B
6765 39                RTS
```

```
10 DATA 02,F0,06,10,06,00,02,F0,
0B,EF,0B,FF
20 FOR N=&H6740 TO &H6745:READ A
$:POKE N,VAL("&H"+A$):NEXT N
30 FOR N=&H6850 TO &H6855:READ A
$:POKE N,VAL("&H"+A$):NEXT N
40 PMODE 0,1:SCREEN 1,0:PCLS
50 CIRCLE(128,96),10
60 IF PEEK(337)=255 THEN 60
70 IF PEEK(135)=94 THEN EXEC&H67
00
80 IF PEEK(135)=10 THEN EXEC&H68
00
90 GOTO 60
```

Other titles from Sunshine

THE WORKING SPECTRUM

David Lawrence 0 946408 00 9 £5.95

THE WORKING DRAGON 32

David Lawrence 0 946408 01 7 £5.95

THE WORKING COMMODORE 64

David Lawrence 0 946408 02 5 £5.95

DRAGON 32 GAMES MASTER

Keith Brain/Steven Brain 0 946408 03 03 £5.95

FUNCTIONAL FORTH for the BBC Computer

Boris Allan 0 946408 04 1 £5.95

COMMODORE 64 MACHINE CODE MASTER

David Lawrence 0 946408 05 X £6.95

SPECTRUM ADVENTURES

Tony Bridge and Roy Carnell 0 946408 07 6 £5.95

THE DRAGON TRAINER

Brian Lloyd 0 946408 09 2 £5.95

Sunshine also publishes

POPULAR COMPUTING WEEKLY

The first weekly magazine for home computer users. Each copy contains Top 10 charts of the best-selling software and books and up-to-the-minute details of the latest games. Other features in the magazine include regular hardware and software reviews, programming hints, computer swap, adventure corner and pages of listings for the Spectrum, Dragon, BBC, Vic 20 and 64, ZX 81 and other popular micros. Only 35p a week, a year's subscription costs £19.95 (£9.98 for six months) in the UK and £37.40 (£18.70 for six months) overseas.

DRAGON USER

The monthly magazine for all users of Dragon microcomputers. Each issue contains reviews of software and peripherals, programming advice for beginners and advanced users, program listings, a technical advisory service and all the latest news related to the Dragon. A year's subscription (12 issues) costs £8.00 in the UK and £14.00 overseas.

For further information contact:
Sunshine
12–13 Little Newport Street
London WC2R 3LD
01-734 3454

Printed in England by Commercial Colour Press, London E.7.

Advanced Sound and Graphics uses a carefully structured approach to show you how to develop routines in your own Dragon programs.

All the major aspects of the sound and graphics capabilities are covered in detail and are fully illustrated. The book takes you from first principles through to bar charts, maps, 3-D projections, movement, animation, direct drawing, screen saving and printing and many other features. Complex sound effects are examined in detail including keyboard sound synthesis, the graphic display of music and the integration of sound and graphics.

In addition to dealing with the operation and applications of the BASIC commands the book explains the internal organisation of the sound and graphics facilities. It also shows you how to use machine code routines to improve your programs.

Keith and Steven Brain are a father and son team and have already published the best selling book Dragon 32 Games Master. They are both regular contributors to Popular Computing Weekly and the monthly magazine Dragon User.

ISBN 0 946408 06 8

£5.95 net