ADVANCED AMSTRAD BASIC

Joe Pritchard

AMSTRAD ADVANCED BASIC

AMSTRAD ADVANCED BASIC

Joe Pritchard

MELBOURNE HOUSE
PUBLISHERS

© 1986 Joe Pritchard

All rights reserved. This book is copyright and no part may be copied or stored by electromagnetic, electronic, photographic, mechanical or any other means whatsoever except as provided by national law. All enquiries should be addressed to the publishers:

Published in United Kingdom by
Melbourne House (Publishers) Ltd
60 High Street, Hampton Wick
Kingston-Upon-Thames
Surrey KT1 4DB

Published in Australia by
Melbourne House (Australia) Pty Ltd
96-100 Tope Street
South Melbourne, Victoria 3205

ISBN 0 86161 202 7

Printed and bound in Great Britain by Short Run Press Ltd, Exeter

Contents

Chapter 1 Graphics and Animation . . . 1

Chapter 2 Sounding Out . . . 77
The Sound commands, envelope design.

Chapter 3 Getting down to BASICs . . . 111
A look at the storage of BASIC programs and variables.

Chapter 4 RSX routines and machine code . . . 141
A collection of RSX routines for easier BASIC programming.

Chapter 5 The CRTC and Screen memory . . . 179
Have complete control of the video facilities of the Amstrad.

Chapter 6 AFTER and EVERY . . . 199

Chapter 7 Some Hints on Programming . . . 209

Appendix 1 ROM Routines Used . . . 239

Appendix 2 BASIC Tokens . . . 241

Index . . . 243

Introduction

This book can be read in a variety of ways; firstly, you can start here, and work your way through to the end. Alternatively, you can dip in at Chapters that interest you, using the Information that you find there for your own programs. The aim of this book is to show what can be achieved on the Amstrad from BASIC. Having said that, we will use machine code on a couple of occasions, but the emphasis will be on the advanced use of Locomotive BASIC. In discussions, I refer to the programmer as 'him'; my apologies to any female programmers. This isn't an attempt to annoy you, it's just that the English language lacks a suitable alternative. (I do not like the word 'one'!!)

So, if you want to get the most from your Amstrad in BASIC, get the machine set up, and off you go . . .

Thanks must go to my publisher for the go ahead and the machine, to Amsoft for their excellent documentation and to my wife for toleration of my anti-social behaviour during the writing of the book!

Joe Pritchard,
Sheffield,
1985

Chapter 1
Graphics and Animation

In this Chapter we'll look at some computer graphics techniques, including some enhanced text printing, the basic principles of animation, and graph drawing amongst others. There will also be several subroutines which you can use in your own programs.

The first area we'll look at is text handling. You'll probably know that as far as text is concerned there are 2 cursors that you can use when printing text; the TEXT CURSOR and the GRAPHICS CURSOR. Normally, text is written to the Text Cursor, which is put at a given position on the screen by the LOCATE command. The text will be written to the current window's Text Cursor in the current window's Text Pen colour. This only allows text to be positioned on the character coordinate system, which depends upon the screen mode, but offers a maximum resolution of 80 by 25 different locations.

The alternative is to write text to the Graphics Cursor, using the TAG command. This allows text to be positioned anywhere on the graphics 'grid' using the MOVE command, and its colour is set by the current graphics Pen colour and the background by the current graphics paper. This allows us to position text on a grid of 640 by 400 pixels. This last statement should be qualified, though; you may not always be able to see a single pixel change.

Vertical resolution, although said to be 0 to 400, is really only 0 to 200. In Mode 1, for example:

```
10 PLOT 200,200
20 PLOT 200,201
```

will plot a single dot on the screen. PLOT 200,202 will then add another dot. Thus, although the quoted resolution is 0 to 400, there are only 200 separately addressable vertical pixels. On the horizontal scale, nominally 640 pixels, there are 640 separately addressable points available only in Mode 2. In Mode 1, PLOT 200,200 and PLOT 201,200 would plot only a single point, thus giving 320 separately addressable points across the screen. In Mode 0 there are 160 separately addressable points across the screen.

However, TAG undoubtedly gives us a means of positioning text with greater precision on the screen. The text is positioned by the MOVE command, the character being printed with its top left corner at the graphics cursor. When in TAG mode, ASCII codes 0 to 31 will print their graphics characters rather than excercise any control function. This has special relevance when printing characters using TAG; any string thus printed must be printed with a ';' after it. If this is not done, the graphics characters for CHR$(10) and CHR$(13) will be added to the string. To see this in action, try the following program:

```
10 MODE 1
20 MOVE 100,100,1
30 TAG
40 PRINT "Hello"
```

Now replace line 40 with:

```
40 PRINT "Hello";
```

The difference is obvious! After a character is printed under TAG, the graphics cursor will be at the top right corner of the last character to be printed. The dimensions of the characters are as follows:

Mode	Vertical	Horizontal
0	16	32
1	16	16
2	16	8

Of course, the above remarks concerning addressable points should be taken into account with regard to this table. This amount of space is taken up by each character printed using TAG, even if the character is a 'narrow' one such as '.' or 'i'. Look in the text of a book, and you will find that different letters occupy different widths on the page. This is known as proportional spacing, and our first subroutine will offer a very simple form of Proportional spacing for use on the Amstrad. It will use TAG to position the letters, but will position the next letter depending upon the width of the current letter. The practical use of such a routine is to allow you to get more characters per line than you

normally would. The number of extra characters depends upon the mode and the text, but can be useful. So, here we go.

Proportional Spacing

The letters chosen for 'special treatment' are as follows. i,l,,', comma,.,:,;. You might like to try adding a few more to this list, but these are the only ones that don't occupy the full width of the space allotted to them. 'o', 'e' and 'c' can be added, but they are slightly 'chopped' by the subroutine. They are included in the listing. In addition, the space allowed for each character is slightly less than usual, which also saves space. It also, unfortunately, trims off the edges of some characters, but the text is always readable, which is what matters. The obvious application for such a routine is in graph labelling, etc. To use the routine, xp% and yp% are set to hold the x and y coordinates of the first character, mp% holds the screen mode, cp% holds the colour. m$ should hold the string to be printed.

```
    1 MODE 2
   10 m$="Now is the time for all good men to come to the aid of the
      party."
   20 mp%=0
   30 cp%=1
   40 xp%=0:yp%=100
   50 GOSUB 20000
   55 TAG:MOVE 0,150:PRINT m$;
   60 END
20000 REM Proportional spacing routine
20010 REM xp%=x position, yp%=y position
20020 REM cp%=colour, mp%=MODE AND m$=string
20030 :
20031 IF mp%=0 THEN inc%=30:prop%=26
20032 IF mp%=1 THEN inc%=15:prop%=12
20040 IF mp%=2 THEN inc%=7:prop%=6
20045 TAG
20050 MOVE xp%,yp%,cp%
20060 FOR i=1 TO LEN(m$)
20070 ch%=inc%
20080 a$=MID$(m$,i,1):IF INSTR("ioec1lt.,!:;' ",a$)<>0 THEN ch%=prop%
20090 PRINT a$;
20100 xp%=xp%+ch%
20110 MOVE xp%,yp%,cp%
20120 NEXT
20130 RETURN
```

The above listing also includes a demonstration text. The routine also allows less space for the space character. Altering the values of 'inc%' and 'prop%' adopted in different screen modes can lead to interesting effects. Note that any text that goes off the right edge of the screen is lost in this subroutine.

To translate between Character Coordinates and Graphics Coordinates, use the below equations. In each case, xg% and yg% are the Graphics Coordinates, and xt% and yt% are the Text Coordinates.

Mode 0
```
xg%=(xt%-1)*32
yg%=(yt%-1)*16:yg%=400-yg%
```

Mode 1
```
xg%=(xt%-1)*16
yg%=(yt%-1)*16:yg%=400-yg%
```

Mode 2
```
xg%=(xt%-1)*8
yg%=(yt%-1)*16:yg%=400-yg%
```

We can also use the TAG function to produce text on the screen that has a 'shadow' effect.

Shadow Effects

A simple effect can be produced whereby text appears to be slightly in front of the background, thus casting a 'shadow' on the background. To make use of this effect, three colours at least are needed: a background colour, black and a foreground colour. The principle of this effect is to set the background, print the text of interest in black on the background, then move the position at which the text is printed, then print the text again in the foreground colour. Of course, it is necessary that the foreground text is printed with transparent paper. The shadow routine prints text to the screen in this way. The routine also includes a demonstration.

```
 1 MEMORY 37999
10 MODE 0
20 cf%=1:cb%=3:mp%=0
30 xp%=10:yp%=10
40 m$="Shadow Mode"
45 PAPER 3:CLS
```

```
   50 GOSUB 20000
   60 END
20000 REM shadow routine
20001 REM cf%=foreground colour, cb%=background
20002 REM xp%, yp% are the text coordinate positions
20003 REM mp%=mode of screen. m$= string
20004 :
20005 IF mp%=0 THEN x%=32
20006 IF mp%=1 THEN x%=16
20007 IF mp%=2 THEN x%=8
20009 FOR i=38000 TO 38005:READ a$:POKE i,VAL("&"+a$):NEXT
20010 x%=(xp%-1)*x%
20020 y%=(400-((yp%-1)*16))
20021 POKE 38001,cb%:CALL 38000
20030 MOVE x%-10,y%+5,5
20040 TAG:PRINT m$;:TAGOFF
20050 PEN cf%
20060 LOCATE xp%,yp%
20070 PRINT CHR$(22);CHR$(1);m$;CHR$(22);CHR$(0)
20079 DATA 3e,00,cd,e4,bb,c9
20080 RETURN
```

In this routine, cf% holds the foreground colour, cb% the background colour of the area of the screen to which the text is to be written, mp% the screen mode, m$ the string and xp% and yp% the Text Coordinate position of the text. The above outline will work on either a CPC 464 or CPC 6128, but to do this a small piece of machine code is used to set the graphics paper, on which the 'black' text is written, to the colour of the background where the text is being written. The graphics paper set like this is NOT transparent, so anything under it will be wiped out. The machine code is set up by line 20009 and line 20021 pokes in the PEN in which the graphics background for the text is to be filled. Once the black 'shadow' is written, the text is rewritten in the foreground colour, using Transparent paper, at the specified text coordinates.

If you have a 6128, the job is a little easier, in that there are commands to set the Graphics Pen and Graphics Paper, and these offer us the possibility of Transparent Graphics Paper. Listed below are the modifications to the above program for use on a 6128 system.

Delete lines 1, 20009, 20079
Alter line 20021 to read:

```
20021 GRAPHICS PAPER cb%
```

To get the Transparent Graphics Paper option, use:

```
20021 GRAPHICS PEN ,1
```

This will also remove the need to set 'cb%' up. Whichever version of the program you use, remember that you will need to reset the graphics paper after using the routine. This can be done by calling the machine code routine after POKEing 38001 with a suitable value for the 464/6128 version, or by a simple GRAPHICS PAPER command for the 6128 version. Note that the machine code is relocatable.

Large Characters

When writing programs that have titles on screens, etc., it's nice to be able to bring attention to these by the use of larger than usual letters. In my book 'Ready Made Machine Language Routines for the Amstrad", I listed some such routines. Here are a couple of BASIC routines to do similar jobs. Note that the |SYMBOL command that we will add in Chapter 4 will be used here.

Double Height

This is the simplest way of getting larger characters, and is particularly useful in Mode 2. In the program, xp% and yp% are the Text Coordinates of the first character of the string, the coordinates referring to the character space occupied by the top half of the letter. cf% is the foreground colour and cb% the background colour. m$ is the string that is to be printed. User Defined Characters 254 and 255 are corrupted by this routine, which works by defining the one character as the top half of a letter and a second character as the bottom half of the letter, and printing them one below the other.

```
    1 MODE 2
    2 cb%=0:cf%=11
    3 xp%=30:yp%=12
    4 m$="Double height text rules OK!"
    5 GOSUB 20000
    6 END
20000 REM double height routine
20010 REM xp%, yp% are the text coordinates
20020 REM m$ is the string to be printed
20030 REM cf% the foreground colour and
20040 REM cb% is the background colour
20050 :
20070 PEN cf%
```

```
20080 PAPER cb%
20090 FOR i%=1 TO LEN(m$)
20100 a$=MID$(m$,i%,1)
20110 |SYMBOL,ASC(a$),36000
20120 SYMBOL
255,PEEK(36000),PEEK(36000),PEEK(36001),PEEK(36001),PEEK(3
6002),PEEK(36002),PEEK(36003),PEEK(36003)
20130 SYMBOL
254,PEEK(36004),PEEK(36004),PEEK(36005),PEEK(36005),PEEK(3
6006), PEEK(36006),PEEK(36007),PEEK(36007)
20140 PRINT CHR$(255);CHR$(10);CHR$(8);CHR$(254);CHR$(11);
20150 NEXT
20160 RETURN
```

Text PEN and PAPER values will be corrupted by the routine. Also, there is no allowance for text over-running the right hand edge of the screen. Note that the |SYMBOL command needs 8 bytes of workspace from which the bytes of the definition can be picked up. I have used 36000 here.

Double Width Characters

This subroutine has the same entry parameters as DBLHGT, but is not as fast. Indeed, it's rather pedestrian. However, the program demonstrates the algorithm and it should easily convert into machine code.

```
    1 MODE 1
    2 cb%=0:cf%=11
    3 xp%=30:yp%=12
    4 m$="Double Width Characters"
    5 GOSUB 20000
    6 END
20000 REM double height routine
20010 REM xp%, yp% are the text coordinates
20020 REM m$ is the string to be printed
20030 REM cf% the foreground colour and
20040 REM cb% is the background colour
20050 :
20060 LOCATE xp%,yp%
20070 PEN cf%
20080 PAPER cb%
20090 FOR i%=1 TO LEN(m$)
20100 a$=MID$(m$,i%,1)
20110 |SYMBOL,ASC(a$),36000
```

```
20119 GOSUB 20180
20120 SYMBOL 255,l%(0),l%(1),l%(2),l%(3),l%(4),l%(5),l%(6),l%(7)
20130 SYMBOL 254,r%(0),r%(1),r%(2),r%(3),r%(4),r%(5),r%(6),r%(7)
20140 PRINT CHR$(255);CHR$(254);
20150 NEXT
20160 RETURN
20170 :
20180 REM adjusts the character definition
20220 FOR row%=0 TO 7
20230 c$=BIN$(PEEK(row%+36000),8)
20235 n%=1:m%=3:GOSUB 20510
20236 l%(row%)=VAL("&X"+b$)
20240 n%=4:m%=7:GOSUB 20510
20250 r%(row%)=VAL("&X"+b$)
20270 NEXT
20280 RETURN
20500 :
20510 REM gets left and right half of
20520 REM the characters
20525 b$=""
20530 FOR j%=n% TO m%
20540 b$=b$+MID$(c$,j%,1)+MID$(c$,j%,1)
20550 NEXT
20560 RETURN
```

Extra Large Characters

The BIGCH routine prints out characters at a massive 8 times larger than normal. Again, machine code would speed matters up somewhat. The routine expects the variable xp%, yp%, m$, cf% and cb% to be set up in the same way as for DBLHGT and DBLWID.

The final subroutine for printing out text strings in different sizes is VARHGT, which prints out characters of varying sizes.

```
    1  MODE 1
    2  cb%=1:cf%=3
    3  xp%=1:yp%=12
    4  m$="BIG!!"
    5  GOSUB 20000
    6  END
20000  REM Big letters routine
20010  REM xp%, yp% are the text coordinates
20020  REM m$ is the string to be printed
20030  REM cf% the foreground colour and
20040  REM cb% is the background colour
```

```
20050 :
20060 LOCATE xp%,yp%
20070 PEN cf%
20080 PAPER cb%
20090 FOR i%=1 TO LEN(m$)
20100 a$=MID$(m$,i%,1)
20110 |SYMBOL,ASC(a$),36000
20119 FOR row%=0 TO 7
20120 c$=BIN$(PEEK(row%+36000),8)
20122 FOR k%=1 TO 8
20123 IF MID$(c$,k%,1)="1" THEN PRINT CHR$(143); ELSE PRINT "
      ";
20124 NEXT
20125 PRINT CHR$(10);:FOR k%=1 TO 8:PRINT CHR$(8);:NEXT
20126 NEXT
20149 LOCATE xp%+(8*i%),yp%
20150 NEXT
20160 RETURN
```

Variable Size Characters

This routine expects the variables cf% and m$ to be set up as before. In addition, sz% holds a value which sets the size of the characters, and offset%, either 1,0 or -1 offers some 'special effects'. offset%=0 with give a straight forward text. Note that in this routine the xp% and yp% variables should hold the position of the first letter of the string in graphics coordinates, and the PEN used will be the current graphics pen.

Again, this routine is slow, and is best used to prepare screens to be loaded in from disc. Obviously, the larger the size selected, the longer the routine will take to draw the characters. The routine is best in Modes 1 and 2 only. The characters are a little distorted in Mode 0. The below listing shows the subroutine, and a demonstration.

```
    1 MODE 2
    2 sz%=3:xp%=100:yp%=100:cf%=1:m$="Hello":offset%=0
    3 GOSUB 20000
    4 END
20000 REM variable size character routine
20010 REM xp% and yp% are graphics
20020 REM coordinates of the text.
20030 REM cf% is the colour, m$ the string
20040 REM sz% is the size factor and offset%
20041 REM gives special effects
```

```
20042 REM BEST IN MODES 1 AND 2. NOT SO GOOD
20043 REM IN MODE 0
20050 :
20051 inc%=8*sz%+2
20052 down%=2*sz%
20060 FOR i%=1 TO LEN(m$)
20065 MOVE xp%+1,yp%-1,cf%
20070 a$=MID$(m$,i%,1)
20080 |SYMBOL,ASC(a$),36000
20090 GOSUB 20200
20091 xp%=xp%+inc%
20093 NEXT
20094 RETURN
20100 :
20200 REM draw a character
20210 top%=yp%
20220 FOR row%=0 TO 7
20230 r$=BIN$(PEEK(36000+row%),8)
20240 FOR j%=1 TO 8
20250 IF MID$(r$,j%,1)="1" THEN c%=1 ELSE c%=0
20255 GOSUB 20320
20260 NEXT
20270 top%=top%-down%
20280 MOVE xp%+1,top%-1
20290 NEXT
20300 RETURN
20310 :
20320 REM plot routine
20330 FOR h%=1 TO sz%
20340 FOR v%=down% TO 1 STEP -1
20350 PLOTR 0,1,c%
20360 NEXT
20370 MOVER 1,-down%+offset%
20380 NEXT
20390 RETURN
```

A lot can be done with text by simply manipulating the bytes that make up the definition of the character. For example, we can print a string 'upside down' by simply getting the definition of each character, inverting it, and then printing a character defined by the new definition. The subroutine, TRANSFORM, prints out strings upside down, back to front, or with the characters on their sides.

TRANSFORM

Here the string is in m$, xp% and yp% the position where the string is to be printed (Text Coordinates), cf% is the PEN colour, cb% the paper colour, and m% defines how the string will be printed.

mode	Description
1	Upside Down
2	Back to Front
3	Sideways
4	Sideways

The two 'sideways' options are different; run the program below to see them! Again, the |SYMBOL command is used to get the character definitions.

```
    1 MODE 1:FOR m%=1 TO 4
    2 xp%=10:yp%=10+m%
    3 cf%=1:cb%=0
    5 m$="Joe is here"
    6 GOSUB 20000
    7 NEXT
    8 END
20000 REM transform routine
20001 REM mode% is the mode of operation
20002 REM xp% and yp% give the position
20003 REM m$ is the string, cf% and cb%
20004 REM the colours.
20005 :
20006 PEN cf%:PAPER cb%
20007 LOCATE xp%,yp%
20010 FOR i%=1 TO LEN(m$)
20020 a$=MID$(m$,i%,1)
20030 |SYMBOL,ASC(a$),36000
20040 ON m% GOSUB 20100,20200,20300,20400
20050 IF m%=2 THEN SYMBOL
255,l%(1),l%(2),l%(3),l%(4),l%(5),l%(6),l%(7),0
20051 IF m%=1 OR m%=3 OR m%=4 THEN SYMBOL
255,l%(1),l%(2),l%(3),l%(4),l%(5),l%(6),l%(7),l%(8)
20060 PRINT CHR$(255);
20070 NEXT
20080 RETURN
```

```
20090 :
20100 REM upside down
20110 j%=1
20120 FOR i=36007 TO 36000 STEP -1
20130 l%(j%)=PEEK(i)
20140 j%=j%+1
20150 NEXT
20160 RETURN
20170 :
20200 REM back to front
20210 FOR row%=1 TO 8
20220 c$=BIN$((PEEK(35999+row%)),8)
20230 d$="":FOR g%=8 TO 1 STEP -1
20240 d$=d$+MID$(c$,g%,1)
20250 NEXT:l%(row%)=VAL("&X"+d$)
20260 NEXT
20270 RETURN
20280 :
20300 REM sideways routine 1
20305 FOR g%=1 TO 8:b$(g%)="":NEXT
20310 FOR row%=1 TO 8
20320 c$=BIN$((PEEK(35999+row%)),8)
20330 FOR column%=8 TO 1 STEP -1
20340 b$(column%)=b$(column%)+MID$(c$,column%,1)
20350 NEXT:NEXT
```

There is an alternative method of getting character definitions for use in such programs that doesn't use machine code but is a little less convenient. This involves printing the character to the screen, then using the POINT command to find out which pixels are in the foreground colour and which are in the background colour. However, this approach has several disadvantages; it requires that a character be printed to the screen, and it is slow.

Well, we've now seen how we can print text in various ways. This would also apply, of course, to User Defined Characters set up by the SYMBOL command. (Not, of course, CHR$(254) or CHR(255) in some of the programs.) Let's now take a look at some advanced ways of using User Defined Characters.

SYMBOL

The standard Locomotive BASIC SYMBOL command allows us to define User Defined Characters. Its use is adequately covered in the manual. However, there is one small 'peculiarity' related to SYMBOL

that some of you may not be aware of.

The SYMBOL AFTER command, which sets the range of characters that are available for the user to change, will not work if you attempt to use it after HIMEM has been changed in some way. This will occur if you've opened a file with OPENIN or OPENOUT, or used the MEMORY command. The moral of this little tale is that you issue SYMBOL AFTER before you do any of these things.

The definition of user defined characters is a tedious job. To try and alleviate the tedium I offer the below program, SYMBOL, which can be used to define a character.

```
 10 REM Character Designer Program
 20 REM Joe Pritchard, September 1985
 30 :
 35 MODE 1
 40 GOSUB 18000
 50 GOSUB 19000
 60 GOSUB 19120
 65
 70 GOSUB 19400
 80 GOSUB 19600
 90 IF m%=5 THEN GOTO 50
100 GOSUB 20000
110 GOTO 70
120 :
130 :
140 :
18000 REM initialise various things
18010 :
18015 WINDOW #2,25,35,10,20:PAPER #2,3:PEN #2,1
18016 CLS #2
18020 DIM l%(8), t%(8), g%(8,8)
18030 SYMBOL 255,0,0,0,0,0,0,0,0
18040 xs%=18:ys%=18
18100 RETURN
18999 :
19000 REM draw the grid
19005 xinc%=8*18:yinc%=xinc%
19010 MOVE 100,100,3
19020 ORIGIN 100,100
19025 DRAW 0,yinc%:DRAW xinc%,yinc%:DRAW xinc%,0:DRAW
        0,0
19030 FOR y%=0 TO yinc% STEP 18
```

```
19040 MOVE 0,y%:DRAW xinc%,y%
19050 NEXT
19060 FOR x%=0 TO xinc% STEP 18
19070 MOVE x%,0:DRAW x%,yinc%
19080 NEXT
19090 RETURN
19100 :
19110 :
19120 REM move cursor and design
19130 REM the character
19140 :
19150 x%=0:y%=18
19155 arx%=1:ary%=1
19160 TAG
19170 :
19175 MOVE x%+2,y%-2,1:PRINT "+";
19180 a$=INKEY$:IF a$="" THEN GOTO 19180
19190 ch%=ASC(a$):MOVE x%+2,y%-2
19195 IF g%(arx%,ary%)=1 THEN PRINT CHR$(143); ELSE
       PRINT CHR$(32);
19200 IF ch%=240 THEN y%=y%+18:ary%=ary%+1:IF ary%=9
       THEN ary%=8:y%=y%-18
19210 IF ch%=241 THEN y%=y%-18:ary%=ary%-1:IF ary%=0
       THEN ary%=1:y%=y%+18
19220 IF ch%=242 THEN x%=x%-18:arx%=arx%-1:IF arx%=0
       THEN arx%=1:x%=x%+18
19230 IF ch%=243 THEN x%=x%+18:arx%=arx%+1:IF arx%=9 THEN
       arx%=8:x%=x%-18
19240 IF ch%=13 THEN g%(arx%,ary%)=1:MOVE x%+2,y%-2:PRINT
       CHR$(143);:x%=x%+18:arx%=arx%+1:GOSUB 19300
19250 IF ch%=32 THEN g%(arx%,ary%)=0:MOVE x%+2,y%-2:PRINT
       CHR$(32);:x%=x%+18:arx%=arx%+1:GOSUB 19300
19251 IF ch%=ASC("Q") OR ch%=ASC("q") THEN RETURN
19252 MOVE x%+2,y%-2:PRINT "+";
19260 GOTO 19180
19270 :
19300 REM fudge routine to take care of line ends
19330 IF arx%=9 AND ary%<>1 THEN
       arx%=1:ary%=ary%-1:x%=0:y%=y%-18:RETURN
19340 IF arx%=9 AND ary%=1 THEN
       ary%=8:arx%=1:x%=0:y%=yinc%:RETURN
19350 RETURN
19360 :
```

```
19370 :
19400 REM routine to put character def.
19410 REM into the array l%()
19420 :
19430 FOR row%=8 TO 1 STEP -1
19440 b$=""
19450 FOR j%=1 TO 8
19460 IF g%(j%,row%)=1 THEN b$=b$+"1" ELSE b$=b$+"0"
19470 NEXT
19480 l%(row%)=VAL("&X"+b$)
19490 NEXT
19500 RETURN
19510 :
19600 REM asks for the mode.
19610 :
19620 PRINT #2,"Mode 1-5"
19630 a$=INKEY$:IF a$="" THEN GOTO 19630
19640 IF INSTR("12345",a$)=0 THEN SOUND 1,200:GOTO 19630
19650 m%=VAL(a$)
19660 IF m%=5 THEN CLS #2:RETURN
19670 PRINT #2:PRINT #2:PRINT #2,"Mode: ";m%
19680 RETURN
20000 REM routine to define character
20010 FOR i%=1 TO 8
20020 t%(i%)=l%(i%)
20030 NEXT
20040 ON m% GOSUB 20100,20200,20300,20400
20050 SYMBOL 255,l%(1),l%(2),l%(3),l%(4),l%(5),l%(6),l%(7),l%(8)
20070 PRINT #2,"Char.: ";CHR$(255)
20075 TAGOFF:LOCATE 1,22:PRINT SPACE$(40);:LOCATE
      1,22:FOR i=1 TO 7:PRINT STR$(l%(i));",";:NEXT:PRINT
      STR$(l%(8)):TAG
20080 RETURN
20090 :
20100 REM normal
20110 j%=1
20120 FOR i%=8 TO 1 STEP -1
20130 l%(j%)=t%(i%)
20140 j%=j%+1
20150 NEXT
20160 RETURN
20170 :
20200 REM upside down
```

```
20210 FOR row%=1 TO 8
20220 c$=BIN$(t%(row%),8)
20230 d$="":FOR g%=8 TO 1 STEP -1
20240 d$=d$+MID$(c$,g%,1)
20250 NEXT:l%(row%)=VAL("&X"+d$)
20260 NEXT
20270 RETURN
20280 :
20300 REM sideways routine 1
20305 FOR g%=1 TO 8:b$(g%)="":NEXT
20310 FOR row%=1 TO 8
20320 c$=BIN$(t%(row%),8)
20330 FOR column%=8 TO 1 STEP -1
20340 b$(column%)=b$(column%)+MID$(c$,column%,1)
20350 NEXT:NEXT
20351 FOR row%=1 TO 8
20352 l%(row%)=VAL("&X"+b$(row%))
20354 NEXT
20360 RETURN
20370 :
20400 REM sideways routine 2
20405 FOR g%=1 TO 8:b$(g%)="":NEXT
20410 FOR row%=1 TO 8
20420 c$=BIN$(t%(row%),8)
20520 FOR column%=1 TO 8
20530 b$(column%)=MID$(c$,column%,1)+b$(column%)
20540 NEXT:NEXT
20550 FOR row%=1 TO 8
20560 l%(9-row%)=VAL("&X"+b$(row%))
20570 NEXT
20580 RETURN
```

Using the program

On running the program you'll be confronted by a Mode 1 screen containing an 8 by 8 grid, and a text window. The '+' in the bottom left of the grid can be moved around by using the 4 arrow keys. Pressing the 'ENTER' key will fill in the square currently occupied by the '+', and this will result in that pixel in the character being defined being set to foreground colour. Pressing 'SPACE' will set the square back to background colour. Pressing 'space' or 'enter' will move the '+' cursor to the right. You can thus see the character being built up in the grid. When you've finished, press the 'Q' key (for 'quit'), and you will be prompted for a 'Mode' to be typed in in the text window. A value of 1 to

4 will print the character just defined and also print out the numbers for a symbol definition. Each of the 1-4 modes prints the character in a different orientation, as you will see if you play around with the program. A mode value of 5 will allow you to continue defining the character. I hope that you will find this program useful.

Multi Coloured Characters

It's very easy to produce characters that contain more than one foreground colour. Such characters are called COMPOSITE CHARACTERS, as they are made up of at least 2 User Defined Characters.

A separate character is defined for each foreground colour that we want to have in the final character.

Thus for a two colour character, we'd need to define two User Defined Characters. An example is shown in Figure 1.1, though the two characters needed to create the character already exist in the character set. The simplest way of creating a two colour character is to use the XOR graphics mode; in the 6128 we could use the GRAPHICS PEN command to produce transparent graphics paper, which is much more useful. However, here I'll just show the XOR method.

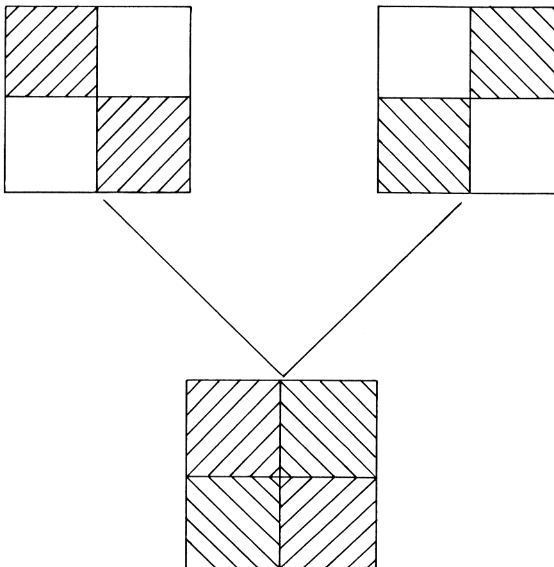


FIGURE 1.1

Figure 1.1 Two Colour Characters

As you can see, there is NO overlap between the different characters used to make up the composite. The reason for this is that any overlap when using XOR can give some rather 'odd' colour effects. (See later in this Chapter.)

To actually get the character on the screen, we can use TAG to print the character at the graphics coordinates and use the XOR graphics mode. The below program demonstrates this technique, printing and moving a two colour character.

```
  10 PRINT CHR$(23)+CHR$(1)
  20 MODE 1
  30 x%=100:y%=100
  40 GOSUB 1000
  50 CALL &BD19
  60 GOSUB 1000
  70 x%=x%
  80 GOTO 40
  90 :
1000 REM routine to print character
1010 TAG
1030 MOVE x%,y%,1
1040 PRINT CHR$(134);
1050 MOVE x%,y%,3
1060 PRINT CHR$(137);
1070 RETURN
```

If you don't want to use graphics coordinates, then you could print the characters at the text coordinates with transparent paper.

Two Dimensional Graphics

We've all done some 2D graphics, even if it's just drawing pictures on a piece of paper! However, in this section of the Chapter we'll see how we can draw various shapes, such as circles, ellipses, etc. The main problem with such complex shapes is that they take time to draw, the main reason for this being the fact that most curved shapes require a calculation involving SIN and COS functions.

These are very time consuming, and various methods have been used to cut down the number of calculations needed, as we shall later see. But before we start looking at some drawing routines, a few words about the graphics screen on the Amstrad.

The Amstrad Graphics Screen

As we've already seen, it's arranged as a 640 * 400 grid, though their are in reality only 200 separately addressable vertical points and the number of separately addressable horizontal points depends upon the screen mode. However, for our purposes, we'll treat the graphics screen as the 640*400 grid.

The Paper and Pen colours used for graphics operations can be set explicitly only in the version of BASIC supplied with the 6128, which has the commands GRAPHICS PEN and GRAPHICS PAPER. On the 464, the graphics PEN colour is set as an additional parameter on the end of a MOVE, DRAW etc. command. However, the graphics pen and paper can be set easily on the 464 with a short machine code routine, such as:

```
10     ORG   36000
20     CP    2
30     RET   NZ
40     LD    A,(IX+0)
50     CALL  &BBE4
60     LD    A,(IX+2)
70     CALL  &BBDE
80     RET
```

Bytes: FE, 02, C0, DD, 7E, 00, CD, E4, BB, DD, 7E, 02, CD, DE, BB, C9

This routine is from my earlier book, 'Ready Made Machine Language Routines for the Amstrad', and is used with a simple CALL command:

```
CALL  address,pen,paper
```

where 'address' is the address of the routine in memory, (it can live anywhere within the memory pool), 'pen' is the desired graphics PEN colour and 'paper' is the desired graphics PAPER colour. The pen colour comes into play immediately, and the paper colour at the next CLG. The 6128 still has some considerable advantages over the 464 in terms of graphics abilities, particularly the ability to produce transparent graphics paper, a facility not present in Version 1.0 BASIC.

It is possible to limit the area of the screen accessed by the various graphics commands by setting up a graphics window. This is done by a variation of the ORIGIN command.

```
ORIGIN ox, oy, left, right, top, bottom
```

where ox and oy are the coordinates of the new graphics origin, and left, right, top and bottom define the limits of the window in graphics coordinates. Any operations from here on will treat ox,oy as the origin rather than the usual 0,0. The commands:

```
10 ORIGIN 300,200
20 CLG
30 PLOT 0,0
```

will plot a point in the centre of the screen, rather than at the bottom left corner as we might expect. One effect of this is that we can have negative 'X' coordinate to the left of the current origin and a negative Y coordinate below it. The below program demonstrates this.
this.

```
10 MODE 1
20 ORIGIN 300,200
25 PLOT 0,0
30 MOVE -100,-100
40 DRAW -100,+100
50 DRAW +100,+100
60 DRAW +100,-100
70 DRAW -100,-100
80 GOTO 80
```

The trick of moving the graphics origin can be a useful one, as we'll see when we come to draw circles, ellipses and curves later in the chapter. However, if you do move it, keep track of its value; the best way to do this is to use a couple of variables so that you can restore it to its original value when necessary.

Only one graphics window is allowed to be active at once, subsequent definitions simply replacing the old one. A nice use of the graphics window is to fill a rectangular block of the screen in with a particular colour; simply define the block of interest as the graphics window, set graphics paper to the desired colour using the above routine, then issue a CLG command. Of course, 6128 programmers can simply use the FILL command.

If you should lose track of the current origin, though, then here is a short machine code routine that will return the current origin in ox% and oy%. These are the ABSOLUTE coordinates of the graphics origin. The routine can be placed anywhere in memory.

```
10      ORG    36000
20     CP    2
30     RET    NZ
```

```
 40     CALL   &BBCC
 50     PUSH   DE
 60     PUSH   HL
 70     LD     L,(IX+0)
 80     LD     H,(IX+1)
 90     POP    DE
100     LD     (HL),E
110     INC    HL
120     LD     (HL),D
130     LD     L,(IX+2)
140     LD     H,(IX+3)
150     POP    DE
160     LD     (HL),E
170     INC    HL
180     LD     (HL),D
190     RET
```

Bytes: FE, 02, C0, CD, CC, BB, D5, E5, DD, 6E, 00, DD, 66, 01, D1, 73, 23, 72, DD, 6E, 02, DD, 66, 03, D1, 73, 23, 72, C9

The routine is used in the following fashion:

```
ox%=0: oy%=0: CALL address, @ox%, @oy%
```

where address is the address of the machine code, ox% the x coordinate of the origin and oy% the y coordinate.

So, after ORIGIN 200,300, the absove line would return with ox%=200 and oy%=300.

The simplest shape to consider drawing is the rectangle. The subroutine 'BOX' offers a general purpose routine for drawing a box on the graphics screen. The below program contains the subroutine and demonstrates it.

```
   10 MODE 2
   20 xp%=RND*600:yp%=RND*300
   30 lx%=RND*100:ly%=RND*50
   40 c%=1:b%=RND*2
   50 GOSUB 20000
   60 GOTO 20
   70 :
   80 :
20000 REM routine to draw boxes
20010 REM xp%, yp% give position of
20020 REM bottom left corner, lx% is
```

```
20030 REM the length and ly% is the
20040 REM height of the box. c% is the
20050 REM colour, and b% sets the border
20060 REM of the box if required.
20070 MOVE xp%,yp%,c%
20080 DRAWR lx%,0
20090 DRAWR 0,ly%
20100 DRAWR -lx%,0
20110 DRAWR 0,-ly%
20120 IF b%=0 THEN RETURN
20130 MOVE xp%+5,yp%+5,c%
20140 DRAWR lx%-10,0
20150 DRAWR 0,ly%-10
20160 DRAWR -(lx%-10),0
20170 DRAWR 0,-(ly%-10)
20180 RETURN
```

Various parameters are set up before the subroutine is called. xp% and yp% set the graphics coordinate position of the bottom left hand corner of the box. lx% holds the length, in graphics pixels, of the box, and ly% its height. c% holds the colour it is to be drawn in. If b%=1 then a second box is drawn just inside the perimeter of the box.

BOX2 is an alternative box drawing program, designed for use with text screens. I've included it here for completion, because it allows the user to signal whether text or graphic coordinates are to be used. A typical application for BOX2 is to provide a 'box' around an area of the screen that is being used for input. Such an input routine is featured in Chapter 7.

```
    1 MODE 2
   10 xp%=100:yp%=100:lx%=20:ly%=5:c%=1:g%=1
   20 GOSUB 20000
   30 END
20000 REM Second box routine
20001 REM xp%, yp%, c% as for the
20002 REM first routine. g% sets
20003 REM whether graphics or text
20004 REM coordinates to be used
20005 :
20010 spt$=CHR$(135)+STRING$(lx%-2,CHR$(131))+CHR$(139)
20020 spa$=CHR$(133)+STRING$(lx%-2," ")+CHR$(138)
20030 spb$=CHR$(141)+STRING$(lx%-2,CHR$(140))+CHR$(142)
20040 IF g%=1 THEN GOTO 20100
20042 LOCATE xp%,yp%:PEN c%
```

```
20044 PRINT spb$;
20046 FOR y%=(yp%-1) TO (yp%-ly%)+1 STEP-1:LOCATE   ,m
        m xp%,y%:PRINT spa$;:NEXT
20048 LOCATE xp%,yp%-ly%+1:PRINT spt$;
20050 RETURN
20060 :
20100 TAG:MOVE xp%,yp%+16,c%
20110 PRINT spb$;
20120 FOR y%=yp%+16 TO ((ly%-1)*16)+yp% STEP 16:MOVE
        xp%,y%+16:PRINT spa$;:NEXT
20130 MOVE xp%,((ly%+1)*16)+yp%:PRINT spt$;
20140 TAGOFF
20150 RETURN
```

xp%, yp%, lx%, ly% and c% are set up as for BOX. However, lx% and ly% are now measured in CHARACTER SQUARES, irrespective of whether the box is being printed at text or graphic coordinates.

g%=1 will cause the routine to assume the coordinates to be seen as graphic coordinates and g%=0 will cause the coordinates to be seen as text coordinates.

In either case, the box is put in the screen using spaces and some of the 'block graphic' characters that are to be found in the Amstrad character set. The result of this is that anything in the area enclosed by the box will be erased when the box is drawn on the screen.

Transformations

The simple box drawn by the 'BOX' routine can provide the start for any 4 sided shape. We can TRANSFORM the rectangle into a different shape by carrying out certain mathematical operations on the coordinate pairs that define each corner of the box. These operations are called transformations, and in this section we'll take a brief look at some of the simpler ones.

A transformation is defined by a MATRIX, which is simply a rectangular array of numbers. This book isn't a mathematical treatise; I couldn't understand it, even if anyone else could! So, we'll simply learn enough about matrices here to allow us to use them. A typical transformation matrix is:

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

although the numbers in such a matrix could be fractional, zero or negative as well as being positive integers. Matrices can be added,

multiplied, etc. just like any other numbers, although the method is a little more involved. If you're interested, I suggest that an 'O' or 'A' level maths text will be useful. All we need to do is to know how to apply such a matrix to a coordinate pair X,Y to give a new pair X',Y'. We say that X,Y is transformed into X',Y'. The matrix we saw above is described as a 2*2 matrix. The coordinates (X,Y) can be described as a 1*2 matrix, and to get X',Y' we simply multiply the two matrices together. The product of two matrices is given by:

$$(X,Y) * \begin{pmatrix} p & q \\ r & s \end{pmatrix} = (p*X+r*Y, q*X+s*Y)$$

$$(X',Y') = (p*X+r*Y , q*X+s*Y)$$

The actual value of X',Y' will thus depend upon the values in the matrix. The simple transformations that we will consider here all have one drawback; that is, they all leave the coordinate (0,0) unaffected. The reason is simple if you think about it- zero multiplied by anything is always zero. All the transformations we will consider will be centered around the coordinate 0,0, therefore. In some cases, this can lead to problems. However, the methods of getting around this are rather beyond the scope of this book, but try to keep this limitation in mind if you use the techniques given below.

The below program, MATRIX, will accept the values for a matrix and apply them to the coordinates of a rectangle, and show the transformed shape. If you are feeling adventurous, you might try other shapes.

```
   10 MODE 1
   20 GOSUB 15000
   25 GOSUB 15210
   30 GOSUB 15130
   40 GOSUB 15280
   50 GOSUB 15310
   60 PRINT #2,"Press any key to go on"
   70 CALL &BB18
   80 CLEAR
   90 GOTO 10
  100 END
14999 :
15000 REM set up the grid and the
15010 REM colours
15020 ORIGIN 320,200,1,640,400,100
15030 MOVE -320,0,3
```

```
15040 DRAW 320,0
15050 MOVE 0,-200
15060 DRAW 0,200
15070 MOVE 0,0,1
15080 PRINT CHR$(23)+CHR$(1)
15110 RETURN
15120 :
15130 REM draw the box using coord%
15140 REM for the start box
15150 MOVE coord%(1),coordy%(1)
15160 FOR i%=2 TO 5
15170 DRAW coord%(i%),coordy%(i%)
15180 NEXT
15190 RETURN
15200 :
15210 REM set arrays up
15211 WINDOW #2,1,40,20,25:PAPER #2,1:PEN #2,3:CLS #2
15220 FOR i%=1 TO 5
15230 READ coord%(i%),coordy%(i%)
15240 coordn%(i%)=coord%(I%):coordny%(i%)=coordy%(i%)
15250 NEXT
15255 DATA 0,50,100,50,100,0,0,0,0,50
15260 RETURN
15270 :
15280 REM input the matrix values
15290 INPUT #2,"P= ",p
15292 INPUT #2,"Q= ",q
15294 INPUT #2,"R= ",r
15296 INPUT #2,"S= ",s
15298 CLS #2
15299 RETURN
15300 :
15310 REM apply the transformation matrix
15320 REM and put the result into
15330 REM coordn%()
15340 FOR i%=1 TO 5
15350 coordn%(i%)=coord%(i%)*p+coordy%(i%)*r
15360 coordny%(i%)=coord%(i%)*q+coordy%(i%)*s
15370 NEXT
15372 :
15373 REM the above applies the matrix
15374 :
15375 REM now delete the original shape
```

```
15376 :
15380 GOSUB 15130
15382 :
15384 REM now redraw the transformed shape
15386 :
15390 MOVE coordn%(1),coordny%(1)
15400 FOR i%=2 TO 5
15410 DRAW coordn%(i%),coordny%(i%)
15420 NEXT
15430 RETURN
```

To use the program, simply type in the appropriate values when asked. Some of the standard transformation matrices are listed below. Once you've tried them out, after the above program to give a shape that doesn't have a coordinate at (0,0), and re-run the program to see the differences.

Identity Matrix

This, when applied to a coordinate, leaves it unchanged.

$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

$$\begin{pmatrix} \text{COS(angle)} & \text{SIN(angle)} \\ \text{-SIN(angle)} & \text{COS(angle)} \end{pmatrix}$$

and for a clockwise rotation around (0,0) the matrix is:

$$\begin{pmatrix} \text{COS(angle)} & \text{-SIN(angle)} \\ \text{SIN(angle)} & \text{COS(angle)} \end{pmatrix}$$

where 'angle' in both cases is the angle through which the shape is being rotated.

Scaling Operations

These allow us to alter the size of the shape by increasing or decreasing the value of X and Y in each coordinate pair.

The Scale Transformation Matrix is:

$$\begin{pmatrix} \text{sizex} & 0 \\ 0 & \text{sizey} \end{pmatrix}$$

'sizex' and 'sizey' are the scaling factors, which determine the size of the transformed shape with relation to the original. If sizex=sizey=1, then the final shape will be the same size as the original. This shouldn't surprise us as we've got the Identity Matrix. The two parameters needn't have the same value; 'sizex' sets the scaling for the X coordinate and 'sizey' the scaling for Y. 'size' values of greater than 1 will cause the transformed shape to be larger than the original. Values less than one will cause it to be smaller.

Reflections

A reflection transform is one which has the effect shown in Figure 1.2. Here we have a reflection in the line on the graph on which all X coordinates are equal to zero.

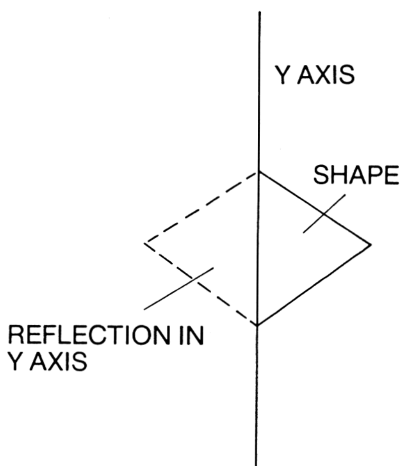


FIGURE 1.2

(Reflection in 'x') This is called the X axis, or the lne 'x=0'. The matrix for this transformation is:

$$\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

You can have a similar reflection in the 'Y' axis, which is given by:

$$\begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix}$$

In addition, you can reflect the shape in any line that you like, such as the line Y=X, on which all the points have equal X and Y coordinates (e.g. 100,100 200,200 300,300 etc.)

Shear Transformations

A shear transformation is one of the type shown in Figure 6.3. There are two types to consider here; the X Shear, which has a Transformation Matrix of:

$$\begin{pmatrix} 1 & shear \\ 0 & 1 \end{pmatrix}$$

and the X Shear, which has the transformation matrix:

$$\begin{pmatrix} 1 & 0 \\ shear & 1 \end{pmatrix}$$

Exactly by how much the shape is distorted by the Shear operation depends upon the value of 'shear'.

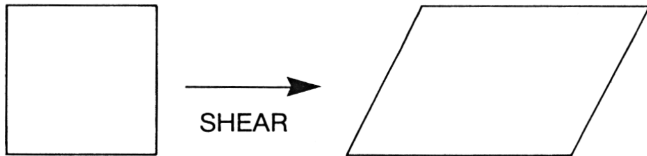


FIGURE 1.3

Shear Transformation

An additional transformation is the TRANSLATION, which is simply a movement away from the original position of the shape as a whole with no distortion. Rather than defining a Transformation Matrix for translation, I simply move the origin and redraw the shape at the new location.

If you perform a translation on a rectangle, then perform another transformation on the transformed shape, the result could have been obtained by multiplying together the two transformation matrices, and simply applying the resultant matrix to the original shape. The product of two matrices is given by:

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} * \begin{pmatrix} p & q \\ r & s \end{pmatrix} = \begin{pmatrix} a*p+b*r & a*q+b*s \\ c*p+d*r & c*q+d*s \end{pmatrix}$$

Transformations are rather handy things to know about, allowing you to generate a variety of shapes from one simple one. For example, a wide variety of 4 sided shapes can be generated from a simple rectangle by the application of suitable transformations. The rotation matrices can be especially useful with complex shapes that need to be rotated. Rather than having to have lists of coordinates for each orientation of the shape we simply recalculate the coordinates when we need them. Of course, there is the problem of time; the SIN and COS operations required take quite a while to perform. The below program, 'PATTERN', simply applies a rotation to a shape repeatedly and draws the result, without erasing the shape previously drawn. You might like to try altering the routine to introduce a second transformation, say a scaling operation, applied after the rotation but before the shape is drawn.

```
   10 MODE 2
   15 GOSUB 15210
   20 FOR angle=0 TO 360 STEP 10
   40 p=COS(angle):q=SIN(angle)
   50 r=-SIN(angle):s=COS(angle)
   60 GOSUB 15310
   70 NEXT
   80 END
15200 :
15210 REM set arrays up
15211 ORIGIN 300,200
15215 DEG
15220 FOR i%=1 TO 5
15230 READ coord%(i%),coordy%(i%)
15240 coordn%(i%)=coord%(I%):coordny%(i%)=coordy%(i%)
15250 NEXT
15255 DATA 0,50,100,50,100,0,0,0,0,50
15260 RETURN
15270 :
15310 REM apply the transformation matrix
15320 REM and put the result into
15330 REM coordn%()
15340 FOR i%=1 TO 5
15350 coordn%(i%)=coord%(i%)*p+coordy%(i%)*r
15360 coordny%(i%)=coord%(i%)*q+coordy%(i%)*s
15370 NEXT
15390 MOVE coordn%(1),coordny%(1)
15400 FOR i%=2 TO 5
15410 DRAW coordn%(i%),coordny%(i%)
15420 NEXT
15430 RETURN
```

That is all we've room for on simple transformations. However, it should prove a useful basis for graphics work.

What about other shapes? Polygons, circles, etc? Well, the main thing to remember about shapes drawn on a computer graphics screen is that they are effectively made up of straight lines. Even if you plot the points that make up the shape, a very time consuming operation, we can still treat the points as being very short straight lines. This leads us to the interesting proposition that a circle can be made up of a great number of straight lines, all very short. And this is the basis of circle drawing. We simply treat the circle as a special polygon.

Drawing Polygons

OK, let's get the joke out of the way first. A polygon is not an escaped parrot! It is simply a shape with several sides. In this section, we'll look at some general techniques with which we can draw shapes of as many sides as we want. We could say that triangles and rectangles are polygons, and it's possible to draw such shapes using the methods we're about to see. But they're so common that we usually draw the 'ad hoc' rather than by using a mathematical equation to derive the shape. In addition, the routines listed here give regular polygons – all their sides are the same length. This is not always desirable when we're drawing rectangles and triangles.

Before we examine some programs, a few general words. The first is about positioning the shape. Most routines of this nature draw the shape around a user specified origin, say 200,300, thus making 200,300 the middle of the shape. Now, there are two ways of drawing this shape. We could leave the graphics origin where it is and add 200 to the X coordinates generated and 300 to the Y coordinates generated, or we could move the graphics origin to the point 200,300. The latter approach is the best. Apart from us not having to add anything to the coordinates, it gives us better shapes; rounding errors will inevitably happen in graphics operations, as we are trying to express real numbers in a limited format. The Firmware of the Amstrad Machines has been written to try and minimise the effects of these errors, and it does this by carrying out it s operations so that rounding errors are all towards the graphics origin. Therefore, to limit the distortion of a given shape, its useful if it can be made symmetrical around the graphics origin. For this reason, all the routines used here will, wherever possible, move the graphics origin to the centre of the shape to be drawn. Thus you might like to use the machine code routine given above to save the current graphics origin before calling any of these routines.

The second general point is again to do with distortion, and is a hardware problem.

Any display will introduce some distortion to the image, and my particular monitor has a tendency to make circles look slightly elliptical. To get around this, where accuracy is needed, a 'fudge factor' is introduced to adjust either the X or Y coordinate. This is a value, in the range 0.8 to 1.2, which is multiplied with the X or Y coordinate to make the circle 'circular' again. For example, if you've got a monitor that distorts in such a way as to make the circle you've drawn 'taller' than it is wide, then you can either reduce each Y coordinate by use of a 'fudge' less than 1, or increase each X coordinate by a 'fudge' greater than 1. Experiment with different values to get the best results. The subroutines listed here DO NOT contain this correction factor; if you need it, simply apply it to each X or Y coordinate accordingly.

POLYGON

This routine draws a polygon of 'n' sides centred on position xp%,yp%. It is of radius 'prad' and in colour c%. Once run, it leaves the graphics origin at xp%,yp%. Any value above 2 will give a recognisable shape, and values above about 25 will give a reasonable circle. The routine is also fairly fast. More accurate circle drawing routines will be given shortly. The below program also includes a demonstration. RUN the program, and press the space bar to see the next shape.

```
    1 MODE 2
    2 xp%=300:yp%=200:c%=1:prad=50
    3 FOR n=1 TO 50
    4 CLS:GOSUB 20000
    5 PRINT n;" sides to this shape"
    6 CALL &BB18
    7 NEXT
20000 REM draws polygons of most types
20030 REM n=number of sides, xp%,yp% position
20040 REM c% colour, prad=radius of shape
20050 REM moves graphics origin
20060 :
20070 pstep=6.28/n
20080 RAD
20090 ORIGIN xp%,yp%
20100 MOVE prad,0,c%
20110 FOR i=0 TO 6.7 STEP pstep
20120 DRAW prad*COS(i),prad*SIN(i)
20130 NEXT
20140 RETURN
```

Although this routine gives a reasonable circle in a fairly short time, we can do better.

Circle Routines

The classic 'equation of a circle' is:

X^2 + Y^2 = R^2

We can rearrange this equation, and get two equations that give values of 'Y' for given values of X and the radius of the circle, R. These equations are:

Y=SQR(R^2 - X^2)
Y=-SQR(R^2 - X^2)

The CIRCLE1 subroutine, listed below with a demonstration routine, uses these two equations to draw the circle in two halves, the top semi-circle being drawn with the equation:

Y=SQR(prad^2-x^2)

and the lower half being drawn with:

Y=-SQR(prad^2-x^2)

xp% and yp% give the position of the circle, c% it s colour and prad giving the radius of the circle. The radius should be integer only. Of course, as well as drawing circles you can also use the routine to draw semi-circles!

```
1 MODE 2
2 prad=INT(RND*50):xp%=RND*300+100:yp%=RND*200+100:c%=1
3 GOSUB 20000
4 GOTO 2
20000 REM draws a circle using the
20010 REM equation of a circle
20020 REM xp%, yp% are position
20030 REM c% is the colour
20040 REM prad is the radius
20050 :
20060 ORIGIN xp%,yp%
20070 MOVE -prad,0,c%
20080 FOR x=-prad TO prad
20090 DRAW x,SQR(prad^2-x^2)
20100 NEXT
20110 FOR x=prad TO -prad STEP -1
20120 DRAW x,-SQR(prad^2-x^2)
```

```
20130 NEXT
20140 RETURN
```

The second technique of drawing circles is based on the use of Trigonometry. Consider Figure 6.4. Here we have a point on a circle defined by the sine and cosine values of the angle, a.

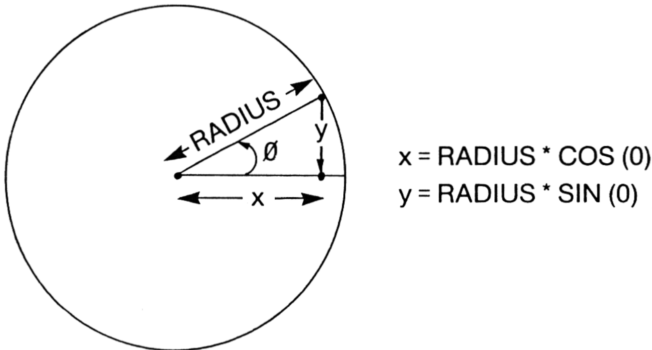


Figure 1.4 Trigonometric view of a circle.

It follows that every point on the perimeter of the circle can be similarly defined, and that we can define the circle by plotting a point at the position specified for 0 degrees, 1 degree, two degrees and so on up to 360 degrees. This method is often called drawing by Angular Increment. The POLYGON routine above worked on this principle. However, we can speed matters up somewhat by using a version of this technique in which the Sine and Cosine calculations are only performed once. All the other sines and cosines required are calculated from these original two. The routine CIRCLE2 draws circles in this way. xp% and yp% set the position, c% the colour and prad the radius.

You can get a smoother circle changing the value of 'n' used from 30 to a higher value. However, this will result in slower operation of the program. It is a fact of life that the smoother you desire the circle, the shorter the lines drawn and the more time needed to draw the complete circle.

```
1 MODE 2
2 prad=INT(RND*50):xp%=RND*
1 300+100:yp%=RND*200+100:c%=
3 GOSUB 20000
4 GOTO 2
```

```
20000 REM draws a circle using the
20010 REM equation of a circle
20020 REM xp%, yp% are position
20030 REM c% is the colour
20040 REM prad is the radius
20050 :
20060 ORIGIN xp%,yp%
20070 MOVE -prad,0,c%
20080 FOR x=-prad TO prad
20090 DRAW x,SQR(prad^2-x^2)
20100 NEXT
20110 FOR x=prad TO -prad STEP -1
20120 DRAW x,-SQR(prad^2-x^2)
20130 NEXT
20140 RETURN
```

That concludes our studies of drawing circles. As a final point, you might like to introduce a line of code into your subroutines that checks whether a given line point is going to be on screen or off screen. If it's off screen then there's little point in drawing it; simply carry on to the next point.

Ellipses

The Ellipse, or oval, is best seen as a 'flattened' circle, and it's by treating it in this way that we can best draw it. A typical ellipse is shown in Figure 1.5.

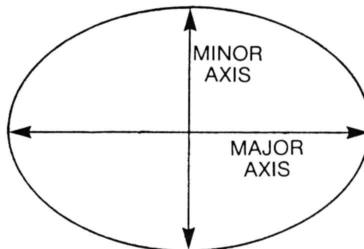


FIGURE 1.5

You will see that an Ellipse is simply a distorted circle. Alternatively, we could call a circle an ellipse in which the sizes of the minor and major axes are the same. Thus, we can draw an ellipse with a subroutine that simply distorts a circle. We've already touched on how to do this when we were discussing how to get rid of screen distortion.

The program ELLIPSE will draw an ellipse, centred on position xp%,yp%, in colour c% and with a major axis radius of 'major' and a minor axis radius of 'minor'.

```
    1 MODE 2
    2 major=INT(RND*50):minor=INT(RND*50):xp%=RND
      *300+100:yp%=RND*200+100:c%=1
    3 GOSUB 20000
    4 GOTO 2
20000 REM draws an ellipse using the
20010 REM sine and cosines
20020 REM xp%, yp% are position
20030 REM c% is the colour
20040 REM major is the radius of the
20042 REM major axis and minor is the
20044 REM radius of the minor axis
20050 :
20060 ORIGIN xp%,yp%
20070 n=30
20080 c=COS(3.14/n) : s=SIN(3.14/n)
20090 RAD
20100 oldc=1 : olds=0
20110 MOVE major*oldc,minor*olds,c%
20120 FOR x=0 TO 2*n
20130 newc=oldc*c-olds*s
20140 news=olds*c+oldc*s
20150 DRAW major*newc,minor*news
20160 oldc=newc : olds=news
20170 NEXT
20180 RETURN
```

The above listing also features a demonstration.

Curves

The simplest curved line to consider is one that is part of the radius of a circle. We can then draw it using a modified version of one of the routines listed already in this Chapter. As a demonstration of this fact, type in and play with the program below:

```
1    MODE 2
2    INPUT "First Angle: ",a
3    INPUT "Second Angle: ",b
4    GOSUB 20060
5    PRINT "Press a key to go on"
```

```
    6   CALL &BB18
    7   GOTO 1
    8   :
20060   REM routine to draw arcs
20070   pstep=360/30
20071   :
20072   REM number of steps is 30
20074   :
20080   DEG
20090   ORIGIN 300,200
20100   MOVE 50*COS(a),50*SIN(a)
20102   :
20104   REM move to position that
20105   REM corresponds to angle a
20106   :
20110   FOR i=a TO b STEP INT(pstep+0.5)
20120   DRAW 50*COS(i),50*SIN(i)
20130   NEXT
20140   RETURN
```

Figure 1.6 shows the angle associated with different parts of the circle for this routine.

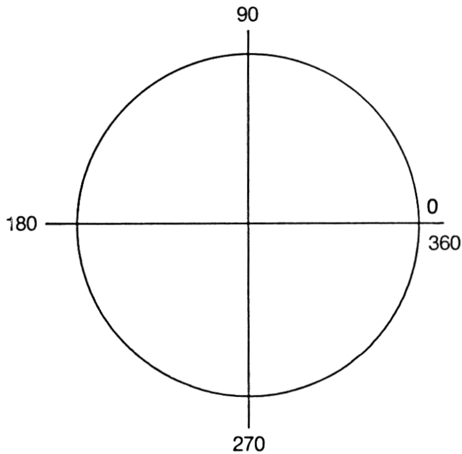


FIGURE 1.6

Circle for Arc Drawing program.

You could draw arcs that are sections of Ellipses by a similar technique, using variables similar to 'major' and 'minor' in the ELLIPSE program. The listing ARC will draw an arc of radius 'prad', in colour c% from starting angle 'a' to finish angle 'b' with the centre of the arc at xp%,yp%. The angles used are those given in Figure 1.6.

```
20000 REM general arc drawing routine
20010 REM xp%,yp% is centre of arc
20020 REM c% is colour
20030 REM prad is radius of arc
20040 REM a is the start angle and
20050 REM b is the finish angle on
20060 REM the circle shown in Fig. 6.6
20065 :
20080 DEG
20090 ORIGIN xp%,yp%
20100  MOVE prad*COS(a),prad*SIN(a),c%
20110 FOR i=a TO b
20120  DRAW prad*COS(i),prad*SIN(i)
20130 NEXT
20140 RETURN
```

The final class of curve we'll examine are those generated by some mathematical function or other. The most obvious of these are things like SIN and COS curves, although SQR will also generate a curve, as will raising numbers to powers, etc. These haven't as much general use as the arc drawing program above. However, it's useful to be aware of them, as they can be used on occasion. In addition, remember that Ellipses, Circles, etc. can all be operated upon by transformation matrices. To do this, the matrix would be applied to every point generated. This allows us to make attractive patterns, similar to that generated by the PATTERN routine, but using curved shapes.

Note that wherever possible, the number of trigonometric operations carried out should be minimised in order to give a faster routine. One way of doing this is to calculate the points that will be needed to draw circles, etc. used by the program before they are needed and store them in arrays. When the circles are needed they can be drawn by simply accessing the arrays holding the points. The below program demonstrates this. The coordinates are held in x() and y(), and are put in the arrays at the start of the program by the subroutine at line 20000. Scaling is done when the points are read back from the array and drawn in the subroutine starting at line 30000.

The main disadvantage of this technique is that more memory is used in dimensioning the arrays than would otherwise be used. However, this isn't too big a problem, with 40 odd k of memory to play with!

```
100 MODE1
110 ON BREAK GOSUB 210
130 PRINT "Press any key to see the"
140 PRINT "Array fill up"
150 CALL &BB18
160 |CRTC,12,0:|CRTC,13,0
170 FOR i=0 TO 2000
180 FOR i=0 TO 100:NEXT
190 a(i)=i
200 NEXT
210 REM Break Key Handler
220 MODE 1
230 END
```

Filling in Shapes.

One thing that is painfully difficult on the Amstrad 464 is to fill in irregular graphics shapes with colour. This has been alleviated by the inclusion in the 6128 BASIC of a FILL command, that allows the shading in of shapes. However, for all you 464 owners out there, here's an introduction to the art of filling in.

The Firmware supports a rather rudimentary Fill function which can be accessed from machine code and which allows the user to fill in rectangular areas of the screen. These routines are called SCR FILL BOX, at &BC44 which operates on columns and rows of the screen, and SCR FLOOD BOX, which operates on screen addresses. These are, though, for rectangular areas of the screen only. We need something a little more sophisticated than this.

Well, we'll start with the basic principles of graphic filling. The aim is kto fill a bounded area with colour, and leave all other areas of the screen alone. Thus in Figure 17, if we told the program to 'fill from x,y', the square would be filled with colour. Telling the program to 'fill from x1,y1' would result in the rest of the screen but NOT the square being filled with colour. This, then, is what we are aiming for; a routine that will fill with colour any area that we want.

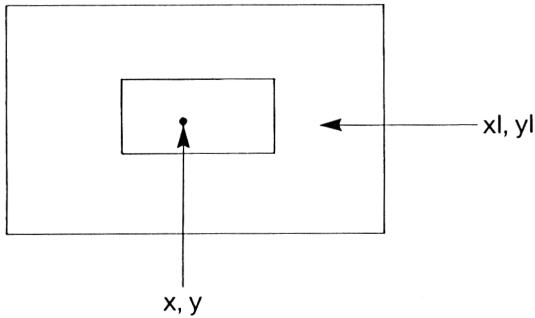


FIGURE 1.7

Plane Graphics.

The edges of an area of screen to be filled are specified by lines called the border, which can be the same or a different colour to that in which we're filling in the shape. In the programs to be discussed here, though, I'll assume a border colour that is different to that of both the background and the fill colour. It should be obvious why the border colour should be different to the background colour! The simplest fill routine to consider is that which fills in a single horizontal or vertical line on the screen between two border colours. The below listing, LFILL, shows a simple demonstration of this.

```
    1 MODE 1
    2 DRAW 0,400,1:MOVE 500,0:DRAW 500,400
    3 ce%=1:cf%=3:xp%=100:yp%=100
    4 GOSUB 22000
    5 END
22000 REM simple line fill in BASIC
22010 REM xp%, yp% is the start pos.
22020 REM cf% is the fill colour, ce%
22030 REM the edge colour
22040 :
22050 off%=1
22060 GOSUB 22140
22070 off%=-1
22080 GOSUB 22140
22090 RETURN
22100 :
22140 REM routine that does it
22150 x%=xp%
```

```
22160 PLOT x%,yp%,cf%
22170 IF TEST (x%+off%,yp%)=ce% THEN RETURN
22180 x%=x%+off%
22185 GOTO 22160
22190 RETURN
```

xp% and yp% specify the start of the fill operation and should be within the line to be filled. cf% sets the colour which is to be used to fill the area, and ce% is the colour that is to be treated as the edge. In this routine, ce% shouldn't be the same as cf%. The algorithm used is very simple; a search is made, first to the right (off%=1), to find the border colour. Any pixel that is not border is plotted in colour cf%. Once the border is reached, a return to xp%,yp% is made and a search is done to the left (off%=-1) in a similar fashion. Thus the whole of the line is filled.

The routine is not very useful, though, as it stands. A line can be filled quite easily in other ways, and most shapes we want to fill are rather more complex! However, a little experiment will allow it to be used to fill simple shapes, such as squares, circles or triangles. As an example, try the below program.

```
  1 MODE 1
 10 MOVE 0,0,1
 20 DRAW 100,100
 30 DRAW 200,0
 40 DRAW 0,0
 50 :
 60 REM draw a triangle in pen 1
 70 :
 80 xp%=100
 90 :
100 REM xp% corresponds to the highest yp%
110 :
120 ce%=1 : cf%=3
130 :
140 REM set the edge colour to 1 and
150 REM set the fill colour to 3
160 :
170 FOR yp%=1 TO 99
180 :
190 REM now loop to fill the shape
200 :
210 GOSUB 22000
```

```
  220 NEXT
  230 END
22000 REM simple line fill in BASIC
22010 REM xp%, yp% is the start pos.
22020 REM cf% is the fill colour, ce%
22030 REM the edge colour
22040 :
22050 off%=1
22060 GOSUB 22140
22070 off%=-1
22080 GOSUB 22140
22090 RETURN
22100 :
22140 REM routine that does it
22150 x%=xp%
22160 PLOT x%,yp%,cf%
22170 IF TEST (x%+off%,yp%)=ce% THEN RETURN
22180 x%=x%+off%
22185 GOTO 22160
22190 RETURN
```

The routine is also a little slow, but this is something that could be circumvented by writing the routine in machine code. xp% is set to correspond to the largest yp% value in the shape, and yp% starts at the lowest yp% in the shape and is taken by the FOR...NEXT loop up to the highest value. Some shapes will therefore need filling in two or three attempts. This still requires us to know the vertical limits of the shape we're filling.

This can be gotten around by calling the subroutine in the following manner:

```
200  GOSUB 22000
210  yp%=yp%+1
220  IF TEST (xp%,yp%+1)<>ce% THEN GOTO 200
```

This will be adequate for simple shapes; again, xp% is set to the x coordinate that corresponds to the largest y coordinate, and we start with yp% set to the lowest y coordinate. To see the type of shape that this approach can fill, try the program LDEMO3. This has to fill the shape in 2 goes.

```
 10 MODE 1
 20 MOVE 0,0,1
 30 DRAW 100,100
 40 DRAW 100,200:DRAW 200,200:DRAW 200,100:DRAW
    300,100:DRAW 300,300:DRAW 400,0:DRAW 0,0
 50 :
 60 REM draw a shape in pen 1
 70 :
 80 xp%=304
 90 :
100 REM xp% corresponds to the highest yp%
110 :
120 yp%=2
130 ce%=1 : cf%=3
140 GOSUB 220
150 yp%=yp%+2
160 IF TEST(xp%,yp%+1)<>ce% THEN GOTO 140
170 REM get here and we're ready to fill rest
180 REM of the shape in in the correct colour
190 xp%=150:yp%=100
200 GOSUB 220
201 yp%=yp%+2
203 IF TEST(xp%,yp%+1)<>ce% THEN GOTO 200
210 END
220 REM simple line fill in BASIC
230 REM xp%, yp% is the start pos.
240 REM cf% is the fill colour, ce%
250 REM the edge colour
260 :
270 off%=2
280 GOSUB 330
290 off%=-2
300 GOSUB 330
310 RETURN
320 :
330 REM routine that does it
340 x%=xp%
350 PLOT x%,yp%,cf%
360 IF TEST (x%+off%,yp%)=ce% THEN RETURN
370 x%=x%+off%
380 GOTO 350
390 RETURN
```

This type of fill method doesn't work too well if there are areas within the shape that are not to be filled in, such as windows in a picture of a house.

Programs that fill any shape with no intervention from the user of the program are, in BASIC, slow and often expensive on memory due to the programming techniques required. Machine code offers the way out, but this type of program is a little beyond the scope of this book. We'll finish this look at fill routines with a program that you can use to 'colour in' graphics screens prior to saving them on to disc or tape. The screen can then be re-loaded into a program. This is often the best way of getting complex graphics screens up quickly, especially in a disc system.

The program FILLF does this.

```
 10 REM fill program for graphics screens
 20 REM designed to be read in from disc
 30 REM Joe Pritchard, 1985
 40 :
 50 MODE 1
 60 :
 70 REM if you want other modes, change
 80 REM xinc% values as follows:
 90 REM mode 0, xinc=4
100 REM mode 2 xinc%=1
110 :
120 xinc%=2:yinc%=2
130 ce%=1:cf%=3
140 :
150 REM default colours are border=1 and
160 REM fill colour=3. Change if you wish
170 :
180 xp%=300:yp%=200
190 x%=0:y%=0
200 :
210 REM this is the main loop of the program
220 REM and it accepts key presses, acts on
230 REM them and prints and moves the cursor
240 :
250 PRINT CHR$(23)+CHR$(1)
260 TAG
270 :
280 MOVE xp%,yp%,ce%
290 x%=xp%:y%=yp%
300 PLOT x%,y%,ce%
```

```
  310 :
  320 REM loop starts here
  330 :
  340 a$=INKEY$:IF a$="" THEN GOTO 340
  350 ch%=ASC(a$)
  360 PLOT x%,y%,ce%
  370 :
  380 IF ch%=240 THEN y%=y%+yinc%
  390 IF ch%=241 THEN y%=y%-yinc%
  400 IF ch%=242 THEN x%=x%-xinc%
  410 IF ch%=243 THEN x%=x%+xinc%
  420 IF ch%=13 THEN GOSUB 19000
  430 IF ch%=70 OR ch%=102 THEN GOSUB 18000
  440 IF ch%=66 OR ch%=98 THEN GOSUB 18080
  442 IF ch%=83 OR ch%=115 THEN GOSUB 19180
  444 IF ch%=76 OR ch%=108 THEN GOSUB 19240
  450 PLOT x%,y%,ce%
  460 GOTO 320
18000 REM input the fill colour.
18010 :
18020 GOSUB 18200
18030 cf%=ax%
18040 RETURN
18050 :
18060 REM enter the border colour
18070 :
18080 GOSUB 18200
18090 ce%=ax%
18100 RETURN
18110 :
18200 REM input routine
18210 TAGOFF
18220 LOCATE 1,1:INPUT"What is the colour: ",ax%
18230 LOCATE 1,1:PRINT "                              ";
18240 TAG:RETURN
18250 :
19000 REM this is the fill routine
19010 REM and uses the subroutine at 20000
19020 REM to fill a single line. This is
19030 REM repeated until a border pixel
19040 REM is reached vertically.
19050 :
19060 xp%=x%:yp%=y%
```

```
19070 TAGOFF
19080 PRINT CHR$(23)+CHR$(0);
19090 :
19100 GOSUB 20000
19110 yp%=yp%+yinc%
19120 IF (TEST (xp%,yp%)<>ce%) AND (yp%<402) THEN GOTO
19100
19130 :
19140 PRINT CHR$(23)+CHR$(1);
19150 TAG
19160 RETURN
19170 :
19180 REM the save routine
19200 GOSUB 19500
19210 SAVE a$,b,&C000,&4000
19220 RETURN
19230 :
19240 REM load routine
19250 GOSUB 19500
19260 MODE 1:TAG
19270 LOAD a$,&C000
19280 RETURN
19290 :
19500 REM input a string
19510 TAGOFF
19520 LOCATE 1,1:INPUT"File Name:  ",a$
19530 LOCATE 1,1:PRINT "                          ";
19540 TAG:RETURN
20000 REM simple line fill in BASIC
20010 REM xp%, yp% is the start pos.
20020 REM cf% is the fill colour, ce%
20030 REM the edge colour
20040 :
20050 off%=xinc%
20060 GOSUB 20110
20070 off%=-xinc%
20080 GOSUB 20110
20090 RETURN
20100 :
20110 REM routine that does it
20120 xd%=xp%
20130 PLOT xd%,yp%,cf%
20140 IF TEST (xd%+off%,yp%)=ce% THEN RETURN
```

```
20145 IF xd%>640 OR xd%<0 THEN RETURN
20150 xd%=xd%+off%
20160 GOTO 20130
20170 RETURN
```

As written, the program is for Mode 1, although details are given in the listing of the alterations needed to make it run in other screen modes. The program offers the following options.

Load a Screen

To do this, press 'L'. Then enter the name of the screen to be loaded and press <ENTER>. The graphics screen will be loaded.

Save a Screen

To do this, press 'S'. Enter the file name under which you want the current screen to be saved, and press <ENTER>.

Change Fill Colour

This alters the colour in which the Fill operation is to be done. The Fill colour shouldn't be the same as the border colour. To use this option, press 'F', then the colour number required, then <ENTER>.

Change Border Colour

This alters the colour used by the fill routine as the border colour, which marks the edge of the area to be filled. Press 'B' then enter the colour number required.

The cursor keys are used to move the 'cursor', a small dot in the current border colour, around the screen. This indicates where the fill operation will start. To fill a particular space with colour, move the dot to within the shape, then press ENTER. The shape will be filled using the line fill routine we've already seen. The fill will extend upwards only, and so the start position should be near the bottom of the area to be filled. Some areas will need a couple of tries to get them totally coloured in. You, the user, provide the 'intelligence' for this fill routine. Once you've coloured in the picture, it can be saved with the save option.

Copying Screen Sections

When putting graphics screens together, a common requirement is the need for a section of the screen to be repeated in two or more parts of the screen. In this section, we'll look at routines that copy areas of the screen.

The simplest approach to this problem is to use TEST() to read pixel colour from the area of the screen being copied and then use PLOT to plot a pixel of the same colour at the 'destination' area of the screen. The subroutine below does this.

```
20000 REM copies an area of the screen
20010 REM xp%,yp% specify block being
20020 REM copied, col% is width, row% is
20030 REM the number of lines, xpd%,xpy%
20040 REM is the destination
20050 :
20055 y%=ypd%
20060 FOR j%=yp% TO (yp%+row%)
20065 x%=xpd%
20070 FOR i%=xp% TO (xp%+col%)
20080 PLOT x%,y%,TEST (i%,j%)
20090 x%=x%+1
20100 NEXT i%
20110 y%=y%+1
20120 NEXT
```

xp%,yp% represent the bottom left hand corner of the block of memory being copied in graphics coordinates. 'col%' holds the length of the area, in pixels, that is to be copied, and 'row%' represents the height of this area. xpd%,ypd% hold the coordinates of the bottom left hand corner of the 'destination' area of the screen.

An alternative copying routine is shown below; this accesses the screen memory, and will only give correct results if the screen hasn't been scrolled. The way in which the screen memory is arranged will be examined later in this book.

```
    1 xp%=1:yp%=1:row%=2:col%=4:xpd%=10:ypd%=10:mp%=1
    2 MODE 1
    3 LOCATE 1,1:PRINT"Hi There!!":LOCATE 1,2:PRINT"Hello"
    4 GOSUB 20000
    5 END
20000 REM copy routine copying
20010 REM from screen memory.
20060 :
20061 REM set up the coordinates to be
20062 REM starting from 0,0, then get
20063 REM the pixel size for each mode
20064 :
20070 xp%=xp%-1:yp%=yp%-1
```

```
20080 xpd%=xpd%-1:ypd%=ypd%-1
20081 IF mp%=0 THEN pix%=4
20082 IF mp%=1 THEN pix%=2
20083 IF mp%=2 THEN pix%=1
20090 :
20091 REM now get the base addresses
20092 REM for each screen location
20093 :
20094 FOR r%=yp% TO yp%+row%
20100 add=&C000+pix%*xp%+80*r%
20110 add2=&C000+pix%*xpd%+80*(ypd%+r%)
20120 :
20130 FOR j%=1 TO 8
20140 FOR i%=0 TO (col%*pix%)-1
20150 POKE (add2+i%),PEEK(add+i%)
20160 NEXT
20170 add=add+2048:add2=add2+2048
20180 NEXT
20185 NEXT
20190 RETURN
```

xp%,yp% hold the top left corner of the area to be copied in text coordinates, from 1 upwards. col% holds the width of the area to be copied, row% the number of text lines to be copied, mp% the screen mode and xpd%,ypd% the text coordinates of the destination area. The above listing includes a quick demonstration. Again, remember that the program produces the correct effect only when the screen hasn't been scrolled.

The final copying routine puts the material read back from the screen into an area of memory, so that it can be put back on the screen later. This routine is called 'COPYM'.

```
1 MEMORY 36999:xp%=1:yp%=1:row%=2:col%=4:mp%=1:opt%=0
2 MODE 1
3 LOCATE 1,1:PRINT"Hi There!!":LOCATE 1,2:PRINT"Hello"
4 GOSUB 20000
5 PRINT"Press a key":CALL &BB18
6 xp%=10:yp%=10:row%=2:col%=4:mp%=1:opt%=1
7 GOSUB 20000
8 END
20000 REM copy routine copying
20010 REM from screen memory.
20020 REM This routine puts the screen
20030 REM contents into a dump space at 37000
```

```
20060 :
20061 REM set up the coordinates to be
20062 REM starting from 0,0, then get
20063 REM the pixel size for each mode
20064 :
20070 xp%=xp%-1:yp%=yp%-1
20081 IF mp%=0 THEN pix%=4
20082 IF mp%=1 THEN pix%=2
20083 IF mp%=2 THEN pix%=1
20090 :
20091 REM now get the base addresses
20092 REM for each screen location
20093 count%=0
20094 FOR r%=yp% TO yp%+row%
20100 add=&C000+pix%*xp%+80*r%
20120 :
20130 FOR j%=1 TO 8
20140 FOR i%=0 TO (col%*pix%)-1
20150 IF opt%=0 THEN POKE (37000+count%),PEEK(add+i%)
      ELSE POKE (add+i%),PEEK(37000+count%)
20155 count%=count%+1
20160 NEXT
20170 add=add+2048
20180 NEXT
20185 NEXT
20190 RETURN
```

xp%,yp%,mp%,row% and col% perform the same function as before. The variable 'opt%' here decides whether data is to be read from the screen or poked to the screen. opt%=0 will read data from the screen into a dump area at address 37000. opt%=1 will read data from this dump and poke it in to screen memory. Note that no error checking is put in to this routine; be careful not to overwrite any of the System Workspace or try and read from incorrect screen locations. All three of these routines can easily be translated into machine code for faster operation.

Simple Animation Techniques

Getting smooth movement on the screen from BASIC is not easy, and in this section we'll examine ways of improving animation. In fact, even machine code routines don't always improve matters, especially if characters are being printed to the screen. This is due to the time needed by the computer to convert the character codes into the bytes needed to produce the correct image on the screen. The only way

around this is to directly access the screen memory from machine code. However, good effects can still be obtained from BASIC, as we'll now see.

At the heart of any routine that is designed to move graphic images around the screen is a piece of code to write the image to the screen, pause, erase it and update the position of the image, then repeat ad infinitum. For example, at the simplest level we could use characters printed at text coordinates, erased by overprinting them with spaces.

This method isn't often used for a couple of reasons.

1. The background is erased; to get around this we would need to copy the area of screen over which the moving character is to pass next and when the character is past that area redraw that section of the screen. This is feasible only in machine code, and then for small characters if smooth movement is to be obtained.

2. Moving one character space at once is a little jerky.

We did use this method in the character designer program, however, where these two considerations weren't important.

To reduce the jerky nature of such movements, we try and move our image in as small steps as possible. In BASIC, this is done by using the TAG command to allow us to move characters around the screen using graphics coordinates rather than text coordinates. This also allows us to use images formed by the use of graphics commands to be moved. In the below program you'll see a few 'dots' moving randomly around the screen, each dot being plotted in one colour and then erased by plotting it in the background colour. This is effectively the same as printing a character and then overprinting with a space; we still zap the background if there's anything there.

```
 10 REM graphics demo in mode 1
 20 REM Joe Pritchard
 30 :
 40 MODE 1
 50 :
 60 DIM x%(4),xo%(4),y%(4),yo%(4)
 70 :
 80 REM set up arrays
 90 :
100 FOR i%=0 TO 4
110 x%(i%)=INT(RND*100)+100
```

```
120 y%(i%)=INT(RND*100)+100
130 NEXT
140 :
150 REM set up start positions
160 :
170 REM main loop starts here
180 :
190 FOR i%=0 TO 4
200 PLOT x%(i%),y%(i%),1
210 xo%(i%)=x%(i%):yo%(i%)=y%(i%)
220 NEXT
230 :
240 REM plot the points and save the
250 REM positions in arrays
260 :
270 FOR i%=0 TO 4
280 x%(i%)=x%(i%)+(INT(RND*3)-1)*2
290 y%(i%)=y%(i%)+(INT(RND*3)-1)*2
300 NEXT
310 :
320 REM new positions into arrays
330 :
340 FOR i%=0 TO 4
350 PLOT xo%(i%),yo%(i%),0
360 NEXT
370 :
380 REM delete points
390 :
400 GOTO 170
```

Note the use of integer variables for speed; we could increase speed a little by getting rid of the REM statements and using multi-statement lines. If you now set the graphics paper to a different colour to '0' and re-run the program , the background will become full of 'holes' as the erase operation will no longer be in the background colour.

There are other ways of erasing characters or plotted points from the graphics screen, and these depend upon us being able to change the way in which anything written to the graphics screen is actually put there. There are 4 Graphics INK MODES, not to be confused with screen modes, that allow us to alter the way in which graphics images are displayed. We'll look at these in more detail shortly, but for now these are:

MODE	Description
0	Force mode. Ink is put on the screen without any modification.
1	XOR Mode. The colour displayed on the screen is obtained by carrying out an exclusive OR operation between the ink in use and what's already on the screen at that point.
2	AND mode. The colour displayed on the screen is obtained by a logical AND operation between the ink and what's there already.
3	OR Mode. The colour displayed on the screen is obtained by a logical OR operation between the ink and what's there already.

The default mode is 0. If you're not sure what the effects of the logical operations are, look them up in the manual. These modes can be set using a 4th parameter with most graphics commands or by:

```
PRINT CHR$(23)+CHR$(mode)
```

where 'mode' is the desired mode. Note that this statement will not work if the TAG system is operative. Instead, you'll get two characters printed to the screen. So, if you're unsure of the status of the TAG command, do a TAGOFF before executing the above.

Our next step in animation uses one of these modes, and we'll later see how the others can also be useful. The XOR mode has a rather unique property, in that printing a shape at the same location, in the same colour, with XOR Ink mode will cause the shape to be

erased! In addition, the background is untouched. There are problems, in that an XORed shape over something else can cause 'odd' effects as the XOR function is carried out between the ink and background, but this ink mode is very useful indeed.

The technique with using XOR then, is to draw the image once, work out the next position at which is to be drawn,draw the shape at the old position then redraw at the new position. To demonstrate this, try the below program.

```
 10 REM demonstration of the XOR mode
 20 REM in moving graphics
 30 :
 40 DIM x(4),y(4),xo(4),yo(4)
 50 :
 60 FOR i%=1 TO 4
 70 READ x(i%),y(i%)
 80 NEXT
 90 :
100 DEG
110 MODE 2
120 ORIGIN 300,200
130 PRINT CHR$(23)+CHR$(1)
140 :
150 MOVE 0,0:FOR j%=1 TO 4:DRAW x(j%),y(j%):NEXT
155 :
160 REM draw the shape first time at new
170 REM position
180 :
190 FOR j%=1 TO 4:xo(J%)=x(j%):yo(j%)=y(j%):x(j%)=x(j%)*1.02:y
    (j%)=y(j%)*1.02:NEXT
195 :
200 REM save original position and then
210 REM scale the shape up a little
220 :
230 MOVE 0,0:FOR j%=1 TO 4:DRAW xo(j%),yo(j%):NEXT
235 :
240 REM erase the original shape by drawing it a second
250 REM time.
260 :
270 GOTO 150
280 DATA 0,50,50,50,50,0,0,0
```

The shape gradually increases in size, the old shape being erased by simply redrawing it. In a similar fashion, we can print characters at the graphics cursor. The routine below uses the cursor keys to move a simple block around the screen, as we might want to do in a game. Note that the repeat rate of the keys have been altered to speed things up.

```
  1 ON BREAK GOSUB 20000
 10 MODE 1
 20 xp%=100:yp%=100:ce%=1
 30 yinc%=4:xinc%=4
 40 a$=CHR$(143)
 50 SPEED KEY 4,4
250 PRINT CHR$(23)+CHR$(1)
260 TAG
270 :
280 MOVE xp%,yp%,ce%
290 x%=xp%:y%=yp%
300 TAG
301 :
302 REM this print statement puts
303 REM the character on the screen for
304 REM the first time
305 :
306 PRINT a$;
310 :
320 REM loop starts here
330 :
340 c$=INKEY$:IF c$="" THEN GOTO 340
350 ch%=ASC(c$)
351 :
352 REM print the character for the
353 REM first time.
354 :
360 MOVE x%,y%:PRINT a$;
370 :
371 REM read the keyboard for cursor keys
372 :
380 IF ch%=240 THEN y%=y%+yinc%
390 IF ch%=241 THEN y%=y%-yinc%
400 IF ch%=242 THEN x%=x%-xinc%
410 IF ch%=243 THEN x%=x%+xinc%
411 :
```

```
  412 REM now print the character for the second time
  413 REM within the loop
  414 :
  450 MOVE x%,y%:PRINT a$;
  455 CALL &BD19
  456 :
  457 REM wait for the frame to be
  458 REM displayed
  459 :
  460 GOTO 320
20000 SPEED KEY 30,2
20010 END
20020 :
20030 REM this sets the key repeat
20040 REM back to normal.
```

The Escape key will finish the program, and reset the keyboard repeat rate. There is some flicker in this routine, which can be reduced by altering the key repeat rate set up by the SPEED KEY statement early in the program. The movement can be made smoother by decreasing the value of 'xinc%' and 'yinc%'. This will also slow down the movement. Increasing the value of these two variables will cause less smooth but faster movement. In addition, flicker will be increased with larger characters or shapes. Again, machine code will speed things up a little.

The call to the ROM routine at &BD19 causes the execution of the program to pause until a frame flyback takes place. This cuts down the flicker by ensuring that the screen is not updated half way through it being displayed.

One point to note is that if you try and update the screen too quickly the character will move and partially 'disappear' at regular intervals. This is due to interaction between the rate at which the hardware refreshes the screen and the rate at which our program attempts to refresh the screen.

The Palette

The range of ink colours available to the programmer is called the palette of colours, just like a painter has a palette. Exactly which PEN holds which colour depends upon the Mode in use. In addition, we can use the INK command to put different colours than usual in a particular PEN. The INK command has the following syntax:

```
INK ink,colour1,colour2
```

where 'colour2' is an optional parameter. The 'ink' parameter, corresponding to the number after a PEN or PAPER command has a value between 0 and 15 and the 'colour' parameters, which select an actual colour, can have a value between 0 and 26. Thus:

```
INK 1,17
PEN 1
```

would result in anything written with PEN 1 appearing on the screen in colour 17.

When an INK command is executed, anything that has been put on the screen in that ink has it s colour changed to that used in the INK command. Thus INK 1,16 will cause anything present on the screen in ink 1 to be displayed in colour 16. This allows us to perform some animation by a technique called Palette Switching. In this method, the inks in a mode are set to background, and a series of images are drawn, each separate image being drawn in a separate ink colour. To animate the image, the ink used for each image is separately set to a non background colour in turn, giving the impression of movement. Of course, there are limitations with this technique, due to the number of colours available in a mode. Palette switching is only really useful, therefore, on the 16 colour Mode 0. The below program demonstrates this technique with the animation of a simple square.

```
 10 REM Palette switched graphics
 20 REM Joe Pritchard
 30 :
 40 MODE 0
 41 ORIGIN 300,200
 42 :
 43 REM set up origin
 44 :
 50 FOR i=1 TO 15
 60 INK i,1
 70 NEXT
 80 :
 90 REM set all inks to background
100 :
110 FOR j%=1 TO 5:READ x(j%),y(j%):NEXT
120 :
130 REM set up shape
140 :
150 FOR i%=1 TO 15
160 MOVE x(1),y(1),i%
```

```
170 FOR f%=2 TO 5:DRAW
    x(f%),y(f%):x(f%)=x(f%)*1.05:y(f%)=y(f%)*1.05
180 NEXT
185 x(1)=x(1)*1.05:y(1)=y(1)*1.05
190 NEXT
200 :
210 REM the above lines draw 15 boxes on
220 REM the screen in different colours.
230 :
240 FOR i=15 TO 1 STEP -1
250 INK i,24
260 CALL &BD19
270 INK i,1
271 :
272 REM set each of the inks in turn to a
273 REM foreground colour to display each
274 REM image in turn then set it to background
275 :
280 NEXT
285 FOR l=0 TO 500:NEXT
286 :
287 REM a delay loop
288 :
290 GOTO 240
300 :
400 DATA -50,-50,-50,50,50,50,50,-50,-50,-50
```

The REM statements in the program explain how it works. The limitations are as follows:

1. There can only be 15 different images on the screen at once, one image for each available colour. Thus an animated diagram can have 15 different 'positions' in it in Mode 0, but only 3 in Mode 1. Palette switching graphics are thus not available in Mode 2.

2. If all the inks are used for animation then none are left for other things.

3. Problems can result if parts of two or more images cross each other on the screen.

Palette switched graphics, therefore, are most use for animating simple, repetitive, diagrams. The fact that all the different images that make up the animation are drawn at once and are displayed by simply

altering the ink in which they're drawn makes the animation very fast and smooth.

Problem (3) above can be a great limitation in graphics, and if you are keen to see it in action, alter the DATA statement in the above program so that the square has a corner at 0,0. This will ensure that two sides overlap. Running the program will now lead to flashing of the image.

The solution to this is to use the OR graphics Ink Mode rather than the Force Mode that we've used so far in Palette Switching. Using this technique reduces the number of separate images available for animation, though, from 15 to 4. The separate images are drawn in inks 1,2,4 and 8. To display the image drawn in colour 2, we simply set ALL the inks that contain a binary 2 to a visible colour. Similarly with the other colours. To see if a colour contains a particular binary value, we could use a line such as:

```
IF (colour AND 4)=0 THEN INK colour,0 ELSE INK colour,24
```

where 'colour' is the ink colour and, in this case, '4' is the binary value we're looking for. The program below demonstrates this technique.

```
 10 REM using OR to switch
 20 REM pallette colours to
 30 REM prevent gaps in images.
 40 :
 50 MODE 0
 60 ORIGIN 300,200
 70 :
 75 FOR i%=1 TO 4:READ colour(i%):NEXT
 80 FOR i=1 TO 15
 90 INK i,1
100 NEXT
105 PRINT CHR$(23)+CHR$(1)
110 :
120 FOR j%=1 TO 5:READ x(j%),y(j%):NEXT
130 :
140 FOR i%=1 TO 4
150 :
160 MOVE x(1),y(1),colour(i%)
170 FOR f%=2 TO 5
180 DRAW x(f%),y(f%)
190 x(f%)=x(f%)*1.05
200 y(f%)=y(f%)*1.05
210 NEXT
```

```
215 NEXT
220 :
230 FOR j%=1 TO 4
240 FOR i%=1 TO 15
250 IF (colour(j%) AND i%)=0 THEN INK i%,0 ELSE INK i%,24
260 NEXT
265 CALL &BD19
270 NEXT
280 GOTO 230
300 DATA 1,2,4,8
310 DATA 0,0,50,0,50,50,0,50,0,0
```

This routine gives more flicker than the previous palette switching technique because of the need to calculate whether a colour contains the appropriate binary number.

In addition to animation, Palette Switching allows us to produce what are called 'Plane Graphics' or 'Sprite Plane Graphics'. This is shown in Figure 1.8.

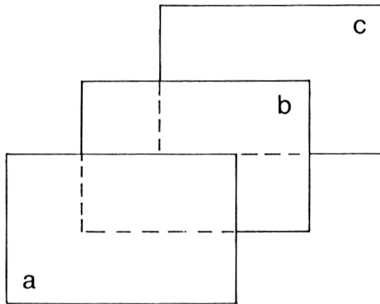


FIGURE 1.8

Filling In

Some machines, such as the MSX computers, have Sprite Plane Graphics as standard, the handling of them being dealt with by the hardware of the computer. The interesting thing about this technique is that a shape moving around on Plane 2 will appear to pass behind any image on Plane 1 but in front of any image present on Plane 3. Plane 1 is said to have a higher priority than Plane 2, which in turn has a higher priority than Plane 3. We can place images on Plane 2 without affecting the images already present on Planes 1 or 3, and we can similarly remove them without damage to the other planes. However, there are some colour limitations, as we are using the colours and the graphics Ink Mode to set the Plane Priority. A system using 5 planes (including a background, thus giving a total of 4 planes

on which images can be drawn) is described in the CPC 6128 manual. Here is a simpler Plane Graphics System for Mode 1, which explains the methods involved. This version has only three different Planes, Plane 1 which is the foreground, Plane 2 which is the midground and Plane 3 which is the background. We can only put images on Planes 1 and 2, therefore, and Plane 1 has the highest priority. Now there is just the problem of getting our images on to the screen.

We use the OR Ink Mode, so let's just look at the pixel colours that will arise on the screen from combinations of the 4 binary patterns available (00 to 11 binary).

Bit pattern in a Pixel	**Plane**	**Colour on the screen**
00	3	PEN 0
01	2	PEN 1
10	1	PEN 2
11	1 and 2	PEN 3

Here, all is clear until bit pattern 11.This will arise when an attempt is made to put an image on Plane 1 at a position which is already occupied by an image on Plane 2. The two bit patterns will be ORed together, thus producing the binary 11 for that particular pixel. Under normal circumstances, therefore, this would result in places where Plane 1 and Plane 2 images coincided having a different colour to Plane 1 or Plane 2. This isn't desirable; the coincident bit pattern, 11, must give rise to the same colour as Plane 1, which is the highest priority plane. Thus, we must redefine INK 3 to be the same colour as Ink 2. We never actually write an image to the screen in Ink 3; we write to Plane 1 in ink 2 and Plane 2 in ink 1.

Let's see this in action. We'll set the background to blue, Plane 2 colour to yellow and Plane 1 colour to red. The commands to do this are:

```
10 INK 0,1 : REM plane 3
20 INK 1,24: REM plane 2
30 INK 2,3 : REM plane 1
```

There is now just Ink 3, which, as we've seen above, must be set to the same colour as Ink 2. So, we can issue:

```
40 INK 3,3
```

to sort this out. To demonstrate this method of drawing graphics, try the below program.

```
 10   INK 0,1
 20   INK 1,24
 30   INK 2,3
 40   INK 3,3
 50   PEN 1
 60   PRINT "First draw Plane 1"
 70   :
 80   REM set up OR graphics mode
 90   :
100   PRINT CHR$(23)+CHR$(1)
110   :
120   REM now draw the Plane 1 image
130   :
140   MOVE 100,100,2
150   DRAW 100,200: DRAW 200,200: DRAW 200,100:
      DRAW 100,100
160   :
170   PRINT "Now press a key for Plane 2"
180   CALL &BB18
190   :
200   MOVE 110,110,1
210   DRAW 110,210: DRAW 210,210: DRAW 210,110:
      DRAW 110,110
220   END
```

So, we can now draw the images on the appropriate planes. You will see from the above program that the drawing of the Plane 2 image above goes behind the Plane 1 image without affecting it in any way. To see what would happen if we didn't set INK 3 to be the same colour as INK 2, alter line 40 to:

```
INK 3,24
```

There now remains the problem of deleting images from one plane without affecting the image on another plane. To do this, let's consider what we must do. Deleting an image on Plane 1 must remove the Plane 1 image but leave any underlying Plane 2 image untouched. Similarly, deleting Plane 2 images must not affect a Plane 1 image. Let's now consider how we might do this. The operation of erasing must have the following effects on the different bit patterns that represent each pixel:

Plane 2 01 erased to Plane 3 00
Plane 1 10 erased to Plane 3 00
Plane 1+2 11 erased to Plane 2 01

It is clear that some sort of logical operation will be applied to each pixel to convert it to the necessary pattern after an image has been erased. The delete operations can be done as follows:

Erase Plane 2 by ANDing the pixel with 2
Erase Plane 1 by ANDing the pixel with 1

Thus we will use the Graphics AND Ink Mode to redraw the image on Plane 2 in graphics pen 2 or use graphics pen 1 to redraw the image on Plane 1. This redraw operation will erase the image on that plane, but it will not damage the image on any other plane. To see how it works:

Plane 2 Erasing.

01 AND 10 = 00

Plane 1 Erasing.

If only a Plane 1 image is at the position concerned:

10 AND 01 = 00

If a Plane 1 and Plane 2 image coincide:

11 AND 01 = 01

thus leaving the Plane 2 image on the screen. Thus we can draw and erase images on the two planes, thus allowing us to produce simple animation in which images pass in front of and behind each other.

We'll finish our look at animation techniques with a simple 'Pseudo Sprite' system. A sprite is a glorified user defined Character which can be positioned on the screen under program control and moved when desired without having to bother about erasing it. The Sprite handler deals with this automatically. In addition, a Sprite System can incorporate means of detecting if two Sprites are overlapping on the screen, and the Sprites can pass in front of or behind each other on the screen, just like our Plane Graphics above, but with more Planes and colours. Some computer Systems, such as

the MSX machines, have Sprite Handling that is controlled by the hardware responsible for the screen display.

Sprite 'collisions' can cause an Interrupt, similar to our AFTER or EVERY interrupt, allowing games programs to be easily written.

Well, the machine code program listed here isn't as complicated as this, but it allows us to move characters around the screen without having to worry about XORing the characters. This is done by the machine code. These 'Pseudo Sprites' are accessed by RSX commands.

```
 10  ; RSX sprites routine
 20  ; using XOR
 30
 40            ORG   35000
 50            LD    BC,table
 60            LD    HL,worksp
 70            CALL  #BCD1
 80            RET
 90
100  temp:     DEFW 0000
110  xcurs:    DEFW 0000
120  ycurs:    DEFW 0000
130  worksp:   DEFS 4
140  table:    DEFW names
150            JP    init
160            JP    sprite
170  names:    DEFM "INI"          ; command names
180            DEFB "T"+128
190            DEFM "SPRIT"
200            DEFB "E"+128
210            DEFB O
220  stable:   DEFS 110            ; sprite info table
230
240  init:     CP    5             ; if not 5 parameters
250            RET   NZ            ; finish
260            LD    B, (IX+8)     ; get sprite number
270            CALL  find
280            CALL  setup
290            LD    IX, (temp)
300            CALL  print
310            RET
320
330  find:     PUSH  IX
```

```
340              LD    IX,stable      ; get base address
350   loop       INC   IX
360              INC   IX
370              INC   IX
380              INC   IX
390              INC   IX
400              INC   IX             ; get the right entry
410              INC   IX
420              INC   IX
430              DJNZ  loop
440              LD    (temp),IX
450              POP   IX
460              RET
470
480   setup:     LD    HL, (temp)
490              LD    A, (IX)        ; transfer char.
500              LD    (HL),A
510              INC   HL
520              INC   HL
530              LD    A,(IX+2)       ; transfer colour
540              LD    (HL),A
550              INC   HL
560              INC   HL
570   coords:    LD    A,(IX+4)       ; transfer y coords.
580              LD    (HL),A
590              INC   HL
600              LD    A,(IX+5)
610              LD    (HL),A
620              INC   HL
630              LD    A,(IX+6)       ; transfer x coordinate.
640              LD    (HL),A
650              INC   HL
660              LD    A,(IX+7)
670              LD    (HL),A
680              RET
690
700   print:     CALL  getcur
710              LD    A,(IX+2)
720              CALL  #BBDE          ; set colour
730              LD    A,1
740              CALL  #BC59          ; set XOR mode
750              LD    E,(IX+6)
760              LD    D,(IX+7)
```

```
770            LD    L,(IX+4)
780            LD    H,(IX+5)      ; get start coordinates
790            CALL  #BBC0
800            LD    A,(IX+0)
810            CALL  #BBFC         ; print character
820            LD    DE,(xcurs)
830            LD    HL,(ycurs)
840            CALL  #BBC0
850            RET
860
870  sprite:   CP    3
880            JR    Z,move        ; move the character
890            CP    4
900            JR    Z,colour      ; change colour
910            CP    5
920            JR    Z,char        ; change character
930            RET
940  move      LD    B,(IX+4)
950            CALL  oldp
960            DEC   IX
970            DEC   IX
980            DEC   IX
990            DEC   IX
1000           LD    HL,(temp)
1010           LD    DE,4
1020           ADD   HL,DE
1030           CALL  coords        ; transfer coords
1040           CALL  newp
1050           RET
1060 colour:   LD    B,(IX+6)
1070           CALL  oldp
1080           LD    A,(IX)
1090           LD    HL,(temp)
1100           INC   HL
1110           INC   HL
1120           LD    (HL),A
1130           CALL  newp
1140           RET
1150 char:     LD    B,(IX+8)
1160           CALL  oldp
1170           LD    A,(IX)
1180           LD    HL,(temp)
1190           LD    (HL),A
```

```
1200            CALL  newp
1210            RET
1220  getcur:   CALL  #BBC6        ; get cursor position
1230            LD    (xcurs),DE
1240            LD    (ycurs),HL
1250            RET
1260  oldp:     CALL  find
1270            PUSH  IX
1280            LD    IX,(temp)
1290            CALL  print
1300            POP   IX
1310            RET
1320  newp:     LD    IX,(temp)
1330            CALL  print
1340            RET
```

The above assembler listing implements the two RSX commands that allow us to use the 'sprites'. Set up the commands by CALL 35000.

At the heart of the program is a 100 byte table, which holds data on up to 10 sprites. The data held in the Table for each sprite is the X and Y position, the Colour of the Sprite and the Character that is to be printed. This character can have any ASCII code you like, and the colour can be any that is allowed in the mode in which the program is being used. The X and Y coordinates specify the position of the Sprite.

To use a Sprite, it must first be initialised, using the |INIT command.

The Syntax is:

|INIT,n,x,y,colour,char

where 'n' is the Sprite Number, 1-10, 'x' and 'y' specify the start position of the sprite, 'colour' specifies the colour in which that Sprite is to be displayed until further notice, and 'char' is the character that is to be displayed as this sprite until further notice. Once executed, this command sets up the entry for Sprite 'n' and will put the character on the screen at the start position and in the specified colour. It is essential that this command is issued before you use the |SPRITE command.

If you want to use the |INIT command after you've had a given sprite running around the screen for some time, then |SPRITE should be used to set the 'x' and 'y' position of the sprite to an area off of the screen. Then an |INIT command can be issued.

If you don't provide the |INIT command with the correct number of parameters the command will be ignored.

The |SPRITE command is the main command added by this program. This allows us to move the character to any position on the screen, and to alter the character printed or the colour in which it's printed. The simplest use of the command is:

```
|SPRITE,n,x,y
```

where 'n' is the Sprite number and 'x' and 'y' specify the new position at which you want Sprite 'n' printed. This will erase the Sprite from it s last position and reprint it at the new position, thus making the job of moving character easier. In addition, the use of machine code speeds things up a little. To see the |SPRITE command in use, moving a character, try the below program. Remember that the RSX commands should have already been initialised.

```
10 MODE 1
20 |INIT,1,100,100,3,65
30 FOR I%=100 TO 400
40 |SPRITE,1,I%,I%
50 NEXT I%
```

You will see the red letter 'A' move up and across the screen. Any flicker present can be lessened by using CALL &BD19, FRAME or a FOR...NEXT delay loop at line 45.

The |SPRITE command saves the current graphics cursor position and the current graphics PEN colour on entry and restores them before returning to BASIC. However, the Graphics Ink Mode is NOT saved, and after a |SPRITE command you'll find that Ink Mode XOR is selected. In practice, though, this isn't too much of a problem.

The second use of |SPRITE is to alter the colour of a given sprite without altering the position of the sprite. The syntax is:

```
|SPRITE,n,x,y,colour
```

where only 'n' and 'colour' are relevant. The values allocated to 'x' and 'y' are used by the program. The sprite will be printed at it s current position in the new colour, and the new colour will be used for subsequent operations with that sprite. 'n' is the Sprite number.

The last use of the |SPRITE command is to change the character that is printed for a given sprite. The syntax is:

```
|SPRITE,n,x,y,colour,char
```

Again, 'n' and 'char' are the only relevant parameters. The new character will be printed at the current position and will be used in subsequent actions with sprite 'n'.

That completes our examination of simple animation techniques.

We'll finish this Chapter by looking at a simple subroutine to take a string of single letter commands referring to graphics operations and act on them, and a quick look at routines for drawing graphs of various types.

DRAW

We've already seen routines similar to this which plays sounds and generates simple rhythms. This routine is the graphics equivalent, accepting a string of simple commands and executing them. It thus provides an alternative to the standard graphics commands and could easily be extended to form a 'Turtle Graphics' program. The routine uses the relative move and draw commands, rather than the absolute ones, so operations are done relative to the point last visited. The unit of measurement used for the Left and Right turns is the degree, and that for the Forward and Backward moves is the Pixel.

The string 'm$' is set to hold any of the following commands, along with any parameter necessary for it s operation. A call is then made to the draw subroutine which will execute each of the commands in the string in turn.

F n.

FORWARD 'n' pixels. Exactly which direction on the screen this will be depends on any angles that have been set up by R or L commands.

At initialisation, F100 would draw a line 100 pixels long towards the top of the screen. R90 followed by F100 would produce a horizontal line 100 pixels long.

B n.

BACKWARD 'n' pixels. Similar to F but moves draws a line backwards relative to the current position.

L n.

LEFT 'n' degrees. This makes a Left Turn of 'n' degrees from the current position. Thus the string:

```
m$="F100L90F100L90F100L90F100"
```

would draw a square with sides of length 100 pixels.

R n.
RIGHT 'n' degrees. This is similar to the L command but does a Right Turn.

P n.
PEN n. This changes the graphics pen to 'n'.

U and D
These allow us to Move as well as DRAW. A 'U' command allows us to move the current location around without drawing a line, and 'D' allows us to start drawing lines again.

```
20000 REM DRAW subroutine and demo
20010 REM Joe Pritchard August 1985
20020 :
20030 :
20050 st$="FBRLPUD1234567890"
20060 dr%=1
20100 :
20110 REM these parameters are set up
20120 REM just the once
20130 :
20140 :
20145 CLS
20146 MOVE 200,200
20150 m$="P1F50R9P2F50R9P3F50R9P4F50R9"
20160 GOSUB 20210
20165 GOTO 20160
20170 END
20180 :
20190 :
20200 :
20210 REM DRAW subroutine
20215 DEG
20230 :
20260 :
20270 FOR i%=1 TO LEN(m$)
20280 a$=MID$(m$,i%,1)
20290 opt%=INSTR(st$,a$)
20300 IF opt%=0 OR opt%>7 THEN ERROR 5
20330 ON opt% GOSUB
20600,20660,20740,20810,20870,20900,20970
20340 NEXT i%
```

```
20350 RETURN
20360 :
20520 :
20530 REM read a number from string
20540 num$="":i%=i%+1
20550 IF i%>LEN(m$) THEN RETURN
20560 a$=MID$(m$,i%,1):IF (a$<="9" AND a$>="0") OR a$="." THEN
      , num$=num$+a$:i%=i%+1:GOTO 20550
20570 i%=i%-1:RETURN
20580 :
20600 REM Forward routine
20610 GOSUB 20530
20620 y%=VAL(num$)
20630 IF dr%=1 THEN DRAWR y%*COS(angle),y%*SIN(angle)
      ELSE MOVER y%*COS(angle),y%*SIN(angle)
20640 RETURN
20650 :
20660 REM Backwards routine
20670 GOSUB 20530
20680 y%=-VAL(num$)
20690 IF dr%=1 THEN DRAWR y%*COS(angle),y%*SIN(angle)
      ELSE MOVER y%*COS(angle),y%*SIN(angle)
20700 RETURN
20710 :
20740 REM Right Turn routine
20750 GOSUB 20530
20760 y%=-VAL(num$)
20761 angle=(angle+y%) MOD 360
20762 RETURN
20800 :
20810 REM Left Turn Routine
20820 GOSUB 20530
20830 y%=VAL(num$)
20840 angle=(angle+y%) MOD 360
20850 RETURN
20860 :
20870 REM pen colour routine
20880 GOSUB 20530
20890 MOVER 0,0,VAL(num$)
20891 RETURN
20892 :
20900 REM Pen Up routine
20910 dr%=0
```

```
20920 RETURN
20930 :
20970 REM Pen Down Routine
20980 dr%=1
20990 RETURN
```

Further features could be added to this routine, such as a REPEAT function, which would allow the repetition of a given sequence of commands a given number of times, and a means of drawing shapes such as squares or circles from one command. The latter could easily be added using routines we've seen already in this Chapter.

We'll finish this Graphics Chapter by looking at ways in which data can be represented on simple graphs, such as Line Graphs, Bar Charts and Pie Charts. These are all useful ways of displaying information on the screen.

Line Graphs

These are the type of graphs we all know from school, in which a line is drawn joining all the points. Examples are trigonometric graphs, such as Y=SIN(X) or Y=COS(X), and graphs describing mathematical equations, such as Y=MX+C. In each case, the Y coordinate is obtained from the X coordinate by a mathematical operation. It's not just a matter of getting the Y values for given X values; the resultant Y values may not be of the correct size to fit on the screen. A graph should be drawn so that as much of the available screen is used as possible. Fitting the graph points on to a graphics screen of a given size is called SCALING, and this is what we'll look at first. There are a few steps to follow to Scale a set of points.

1. Find the maximum value of X, Xm, and the maximum value of Y, Ym.

2. Calculate the scale factor, scf, for each axis by calculating scfx=X/Xm and scfy=Y/Ym. 'Y' is the largest value in the Y axis, and 'X' is the largest value of the X axis.

3. Multiply each X coordinate by scfx, and each Y coordinate by scfy.

In addition, an OFFSET is sometimes required. This is a value that is added to or subtracted from the Y coordinate values AFTER they've been scaled. It serves to move the plotted points up and down the Y axis. One typical use of this is when we have a function, such as SIN(), that returns negative values over part of it s range. The offset is added

to ensure that all the points drawn are above 0. An alternative to this, of course, is to set the graphics origin appropriately – say with y=200.

The first prerequisite here is that we have the coordinates already in the machine. The most efficient way to do this is to store them in Arrays. The listing below shows how we can set up arrays containing a series of points to be plotted and then scale and plot them. Note how the origin is moved so as to avoid the need to add an offset value. The listing is self documenting.

```
 10 REM Demonstration of scaling of a line
 20 REM graph. Joe Pritchard, 1985
 30 :
 40 DIM x(360),y(360)
 50 DIM x%(360),y%(360)
 60 :
 70 REM dimension the arrays
 80 :
 90 Y=200:X=640
100 :
110 REM largest Y coordinate is 200, as we'll
120 REM have moved the origin to 0,200 thus
130 REM allowing us to plot points without
140 REM adding an offset
150 :
160 ORIGIN 0,200
170 DEG
180 FOR i%=0 TO 360
190 x(i%)=i%
200 y(i%)=SIN(i%)
210 NEXT i%
220 :
230 REM Lines 160-210 set the origin, set degrees
240 REM and set up the arrays with the data. x(n)
250 REM is set for the sake of example: you could
260 REM get around having to use x() and x%()
270 :
280 xm=0
290 FOR i%=0 TO 360
300 IF ABS(x(i%))>xm THEN xm=ABS(x(i%))
310 NEXT i%
320 :
330 REM get the largeset value of x into xm
340 :
```

```
350 ym=0
360 FOR i%=0 TO 360
370 IF ABS(y(i%))>ym THEN ym=ABS(y(i%))
380 NEXT
390 :
400 REM now get the largest y value into ym
410 :
420 xscf=640/xm
430 yscf=200/ym
440 :
450 REM work out the scaling factors.
460 :
470 FOR i%=0 TO 360
480 x%(i%)=INT(x(i%)*xscf)
490 y%(i%)=INT(y(i%)*yscf)
500 NEXT i%
510 :
520 REM scale the points and store in arrays
530 :
540 MOVE 0,0,1
550 FOR i%=0 TO 360
560 DRAW x%(i%),y%(i%)
570 NEXT i%
580 :
590 REM now move to the origin and draw the
600 REM graph that we want, using the full amount
610 REM of the screen that is available to us
```

There is, of course, the matter of drawing and labelling the axes. Drawing them is fairly straight forward; the Envelope designer program will show how this could be done. However, labelling is a bigger problem; sometimes the numbers are simply too large to fit on to the graph. This can be partially alleviated by using the graphics cursor to position the numbers, or by redefining some of the user defined characters as 'thin' numbers. This is a particular problem in Mode 0. As an alternative to this, you might consider the following approach.

The graph is drawn, and a 'cursor' is positioned on the screen. This cursor can be moved around using the arrow keys and can thus be positioned at any point on the screen. The X and Y position of the cursor will be known, and this, in conjunction with the scale factors used, can be used to calculate the 'real' values represented by the graph. These can then be displayed in a text window. The actual program needed to do this is fairly simple, so I leave it to you!

Bar Charts

A second method of displaying information is the Bar Chart, in which the graph consists of a series of vertical 'bars'. The basic principles of drawing a bar chart are as follows.

1. The number of X values gives the number of bars that the chart will have.

2. Use a routine similar to that above to get the largest Y value, and then use this to produce a scaling factor for the Y values. Apply the scaling factor to the Y values. This will give you the height of the 'bar' that is to be drawn on the screen.

3. Now draw the bar. There aren't many quick ways of doing this, as we usually want the bar to be filled in with colour. From BASIC, the best way is to use the height of the bar and it s width to define a graphics window, clear this to the graphics paper desired, then reset the graphics window to it s original size. This does mean keeping the original size of window, origin and graphics paper in variables, but this isn't a problem. If you've a 6128 system you can draw the box and then use the FILL command to fill it in. An alternative that might be useful is to use CHR$(143) to print the bar on the screen, using the TAG option to position the bar.

Pie Charts

We'll finish off now with a simple routine to draw Pie Charts. These are circular diagrams that have been split into sectors, the size of the perimeter of each such sector representing the size of the parameter being graphed.

The principle of drawing such a chart is fairly simple; we take the values to be graphed and scale them so that they'll fit into a 360 degree circle. This gives us all the data we need to draw a circle with the sectors marked out. We simply draw the circle as a series of arcs, as was shown earlier in this Chapter. The below program shows this in action, and can easily be written as a subroutine to fit in to your own programs. Although I've only used 5 items of data, you could use more. Be warned that the more data you have, the larger the circle radius needs to be to show the different sectors clearly. Again, the routine is well REMmed so you can follow and modify it.

```
 10 REM Pie Chart Drawing Demonstration.
 20 REM Joe Pritchard, 1985
 30 :
 40 FOR i=1 TO 5
 50 READ y(i)
 60 NEXT
 70 :
 80 REM read data in from data statement
 90 REM at line 30000
100 REM could easily initialise array from
110 REM input statements or a data file
120 REM if you did and used more than 10
130 REM items, don't forget to DIM arrays
140 :
150 ytot=0
160 FOR i=1 TO 5
170 ytot=ytot+y(i)
180 NEXT
190 :
200 REM totalise the values so that we can
210 REM work out the proportion that each
220 REM Y data value is of the whole
230 :
240 FOR i%=1 TO 5
250 y%(i%)=INT(((y(i%)/ytot)*100)+0.5)*3.6
260 NEXT
270 :
280 REM work out the percentage that each y
290 REM data value represents and then
300 REM calculate the angle that it represents
310 REM as part of a circle
320 :
321 FOR i%=1 TO 5
322 y%(i%)=y%(i%)+y%(i%-1)
323 NEXT
324 :
325 REM now add up the values to get the
326 REM angles that are used for each arc
327 :
330 xp%=300:yp%=200:prad=100:c%=1
350 :
360 REM set up the parameters for the
370 REM 'circle' that will be drawn out
```

```
  380 REM of the various segments Centre it on
  390 REM 300,200, in colour 1, with a
  400 REM radius of 100.
  430 :
  440 FOR i%=1 TO 5
  450 a=y%(i%-1):b=y%(i%)
  460 GOSUB 20000
  470 NEXT
20000 REM Modified arc drawing routine
20010 REM xp%,yp% is centre of arc
20020 REM c% is colour
20030 REM prad is radius of arc
20040 REM a is the start angle and
20050 REM b is the finish angle on
20060 REM the circle shown in Fig. 6.6
20065 :
20080 DEG
20090 ORIGIN xp%,yp%:MOVE 0,0
20100 DRAW prad*COS(a),prad*SIN(a),c%
20110 FOR i=a TO b
20120 DRAW prad*COS(i),prad*SIN(i)
20130 NEXT
20140 RETURN
30000 DATA 10,10,30,40,50
```

You might like to add labelling facilities to the routine, and colouring facilities. This labelling can be easily done on larger charts with a small number of sectors by using TAG to print a code letter in the sector. This isn't so easy with smaller sectors.

Chapter 2
Sounding Out

In this Chapter we'll consider sound. I've assumed that you are already conversant with the contents of the manual on this subject, so you won't find much in the Chapter on the basics of sound production. Instead, we'll concentrate on the more advanced aspects of Sound production, including Envelope Design, synchronization of sound on different Channels, sound effects and directly accessing the Sound Generator hardware. We'll round off with a few useful subroutines and programs for sound.

The SOUND command

Just a quick review of the SOUND command. The full Syntax is:

SOUND channel,tone,duration,volume,ENV,ENT,noise

Channel

This parameter is an 8 bit value which can be viewed on a bitwise basis. We'll look at the details of synchronization, Holding and Release of Channels later in this Chapter.

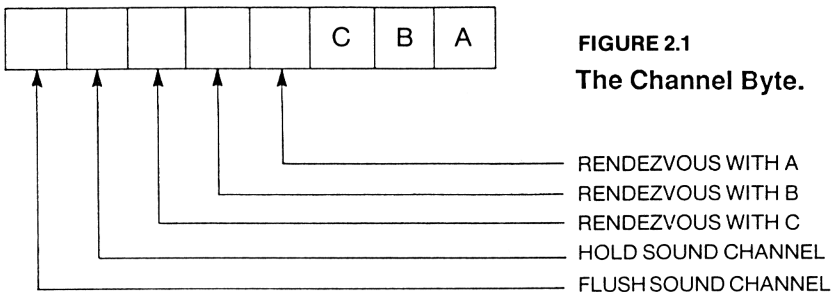


FIGURE 2.1
The Channel Byte.

This parameter is compulsory.

Tone
This defines the period of the tone played, and hence it s pitch. If you have a Tone Envelope in use as well, then obviously the tone will vary as the sound is played. Legal values are between 0 and 4095, 4095 being the lowest pitch.

Duration
This is the period of time, in units of 0.01 seconds, for which the Channel is active. The default is a duration of 20 units, or 1/5 second. In this role, the legal values are between 0 and 32767.

If the parameter is used in conjunction with Envelope commands, then there are two special cases. If the Duration is set to zero, then the duration will depend upon the duration of the Envelope (see later in Chapter). The Channel will simply play the active Envelope, then go silent.

If a negative value is given, say -10, then the envelope will be played 10 times. Similarly, -4 would play the envelope 4 times.

Volume
This is the starting Volume of the sound being played on the Channel. Legal values are between 0 and 15, 15 being the loudest note available. If you omit this parameter, then we have a volume of 12.

ENV and ENT
These are the Amplitude and Tone Envelopes, which we'll look at later in the Chapter. In both cases, legal values are in the range 1-15.

Noise
The last parameter of the SOUND command is that defining the pitch of any noise being played on the Channel. Legal values are in the range 1 to 31, with 31 giving noise of the lowest 'pitch' and 1 being the highest pitch.

Noise is typically used in programs to provide sound effects of various types. The noise produced is often called 'White' or 'Random' noise, and if you want to hear an example try the below line:

```
SOUND 1,0,duration,volume,0,0,noise
```

where 'noise' is the pitch of the noise required, 'duration' and 'volume' are as given above. The 'tone' parameter has been set to zero above, but if you set this parameter to another value then you would get a

mixture of tone and noise being played on Channel A. The below program will play the range of noise 'pitches' available on the Amstrad.

```
10 FOR pitch=0 TO 31
20 SOUND 1,0,10,15,0,0,pitch
30 NEXT pitch
```

Noise can be used to add a variety of sound effects to your programs, with or without the use of Envelopes. You might like to try the two programs below to get an idea of the potential of even simple routines.

```
10 REM rain on a roof??
20 J%=RND*50
30 FOR I%=0 TO J%:NEXT I%
50 SOUND 1,0,1,RND*15,0,0,RND*31
60 GOTO 20
```

```
10 REM running footsteps for a game?
20 J%=200
30 FOR I%=0 TO J%:NEXT I%
50 SOUND 1,0,2,15,0,0,31
60 GOTO 20
```

Some experimentation with these two routines – e.g. adjusting the value of J% or the noise pitch and duration, or introducing some tone – would probably give other effects. We'll see some more sound effects when we consider envelopes later. As a final example, try the below 'explosion'.

```
FOR I%=15 TO 0 STEP -1:SOUND 1,0,5,I%,0,0,10: NEXT I%
```

Synchronization of Channels

You've probably already 'beeped' out tunes on the Amstrad; if not you might like to try the 'Notes' program later in this Chapter. But a common requirement in music and sound effects is for notes to be played simultaneously on different channels, each channel playing a different note. Synchronising two or three channels in this way isn't as hard as it sounds, so we'll now examine how we can do it.

For example, say we want to play to tones, of period 200 and 300 on Channels A and B at the same time. A will be Synchronised with B,

and vice versa. Amsoft use the term 'RENDEZVOUS' when discussing synchronisation of notes, and an alternative way of putting this is that 'A will rendezvous with B'. Once we tell two or more Channels to rendezvous in this way, the only time that we'll get any sound out of one channel is when the other channels (with which it is to rendezvous) have got a sound in their queue.

Rendezvous can be applied to Noise as well as tone, and to Enveloped or Non-Enveloped sounds.

So, how is it done? We use what I call the SYNCHRONISATION BITS of the 'channel' parameter, Bits 3, 4 and 5. If we want to rendezvous a given channel, with, say Channel A, we simply set Bit 3 to 1 as well as setting the appropriate Channel Select bits (Bits 0 to 2). Telling a Channel to rendezvous with itself is, by the way, a little pointless! So, back to our original problem. Let's play the tone with period 200 on Channel A. We need to get this Channel to rendezvous with Channel B. The line of BASIC to do this is:

```
10 SOUND &X00010001,200
```

I've expressed it in binary rather than decimal for clarity. The Channel B rendezvous bit, Bit 4, and the Channel A select bit, Bit 0, have both been set to '1'. In a similar fashion, we can issue the below command to Channel B:

```
20 SOUND &X00001010,300
```

Now type in the lines without line numbers. Note how no sound is heard until after BOTH the commands are entered. An interesting effect is noted if we use:

```
10 SOUND &X00010001,200,1000
20 SOUND &X00001010,202,1000
```

A 'beat' note can be heard, corresponding to the difference in frequency between the two tone being played.

If we wanted to synchronise the activity of all three channels, then we would use the following 'channel' parameters.

Channel A	&X00110001	Sync with B+C
Channel B	&X00101010	Sync with A+C
Channel C	&X00011100	Sync with A+B

One practical application of the rendezvous is the ability to play musical chords. This isn't a music textbook, so all you'll get here are the very basics.

A chord is a pleasant sounding combination of musical notes, often 3. There are many different chords possible, but here are a couple that might be of interest to you.

M=Major, m=Minor.

Chord	**Notes**	**Tone Period Values**
CM7	C,E,A#	239, 190, 134
Am	A,C,E	142, 239, 190
AM7	A,C#,G	142, 225, 159
EM7	E,G#,D	190, 150, 213

To play them, just play all three notes at the same time.

Hold

It is possible to set up a sound, but not play it until a set time in the future. The HOLD function on a given Channel is set by bit 6 of the 'channel' parameter for that Channel. If set to 1, then the sound is placed in the Sound Queue but not played at once. Try:

```
SOUND &X10000001,200
```

No sound is heard until a RELEASE command is used. In this case, RELEASE 1 will allow the tone to be heard. The parameter of RELEASE is a 3 bit binary number, which specifies which channels should be RELEASEd.

Channel A
Channel B
Channel C

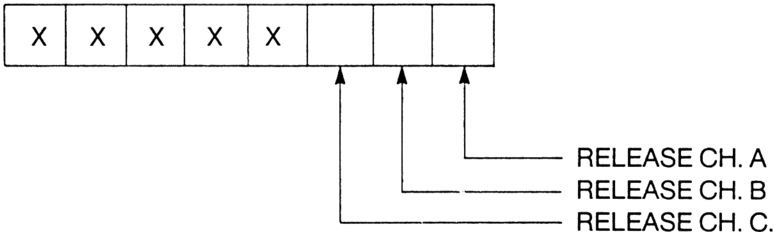


FIGURE 2.2

As with the 'channel' parameter, these are additive. i.e. to RELEASE any sounds in the Sound Queue of Channels A and B we'd simply execute a

```
RELEASE 3
```

command. To see RELEASE in action, try the below program.

```
10 SOUND &X01000001,200
20 SOUND 1,300
30 SOUND 1,400
```

RUN it. No sound will be produced. Now type in RELEASE 1. This will allow all three notes to be heard, one after another. Any sounds added to a sound queue after a sound that has been flagged as 'Held' by the 'channel' parameter will also be held.

Flush

Bit 7 of the 'channel' parameter is used to terminate sound production on a given Channel. This has the effect of turning off any sound being produced, clearing the queue, and playing the sound with the 'Flush' bit set immediately. To see it in action, try:

```
10 SOUND 1,200,10000
20 a$="":IF a$="" THEN GOTO 20
30 SOUND &X10000001,400
```

RUN the program; once a key is pressed you'll hear a change in tone then silence.

To see what would happen without the 'Flush' bit set, replace line 30 with:

```
30 SOUND 1,400
```

The 'Flush' bit thus gives us a kind of 'priority' sound, which will have precedence over anything else in the Sound Queue. To simply silence Channel A, for example, execute:

```
SOUND 129,0
```

Sound Queues

The computer 'stacks up' sound commands that it has received for each Channel in what is called a Sound Queue. The Programmable

Sound Generator deals with the production of sounds without any interference from the CPU once it has been set going, but requires updates from the CPU when the sound needs altering, as when Software Amplitude or Tone Envelopes are being processed. As sounds are produced in 'real time', the chances are that a request for sound will come before a previous request has been fully dealt with. The second sound is simply remembered and when the first sound has finished the second can be handled.

The practical upshot of this is that when a lot of sounds are being produced, one after another, on the same channel, there is often a pause in the program execution while the Firmware tries to put a new sound into an already full sound queue! Processing waits until there is a space in the queue. This is not a desirable occurrence, and Amstrad BASIC provides a means around such problems, as we'll soon see.

The status of each Sound Queue can be evaluated using the SQ() function. This returns an 8 bit value which is decoded in the below fashion.

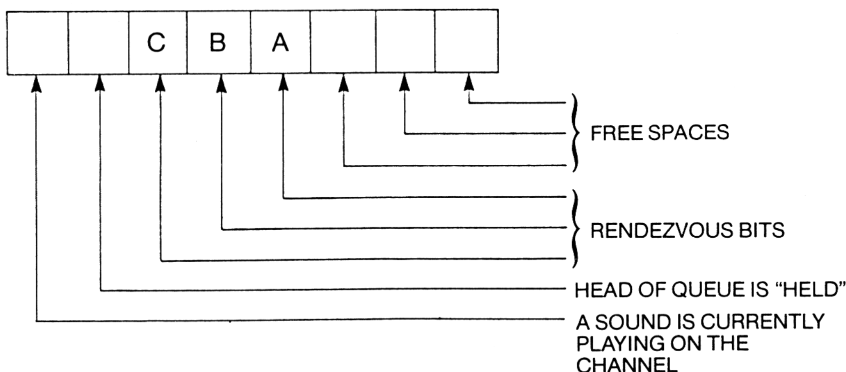


FIGURE 2.3

Bits 0 to 2 return the number of free spaces in the Sound Queue of that Channel at that moment in time. The other bits are flags, if set to '1' indicating that the fact is true and if set to '0' indicating that the fact is false. Bit 3 indicates that the sound at the head of the queue is to rendezvous with Channel A, Bit 4 to rendezvous with Channel B and Bit 5 to rendezvous with Channel C.

Bit 6 indicates that the sound at the front of the Sound Queue is 'Held'. Bit 7 is set to '1' if a sound is currently being generated on this Channel.

Due to this bitwise arrangement, if we want to isolate certain information from the result returned by SQ we use the bitwise AND command. Thus to return the number of free spaces in Channel A we'd execute a:

```
free=(SQ(1) AND &X00000111)
```

To see the SQ function in use, try the below program.

```
10 chann=1
20 FOR I=0 TO 30
50 SOUND chann,200
60 x=SQ(1)
70 y=SQ(2)
80 PRINT chann,x,y
90 NEXT I
```

RUN the program, and you'll see the two channel statuses printed out. In addition, the sound carries on after the program is finished. Now add the following two lines:

```
30 IF (x AND &X00000111)=0 THEN chann=2
40 IF (y AND &X00000111)=0 THEN chann=1
```

These lines simply monitor the Queues of Channels A and B and when there are no spaces left in the queue the 'chann' variable is altered to switch sound output to the other channel. At some points, therefore, there will be sound on both A and B.

Obviously, you could also use the SQ function to only issue a SOUND command when there is room is the queue of the appropriate Channel to play it.

The only values that make sense to the SQ function are 1,2 and 4. If a value such as 5 is used, then information about only one channel will be returned. In this case, 5 is equal to 1+4, and the information on the lowest numbered Channel, in this case Channel A, will be returned.

```
ON SQ(n) GOSUB m
```

This command allows a subroutine at line m to be entered when a free space in the Sound Queue of Channel n appears. It can thus be used to only send a sound to a channel when a free space will be in the sound queue, thus ensuring that the program doesn't have to 'wait' for

a free space to arise. It can thus be used to play continuous music, for example, in a games program. There is only one point to note about this command; it is deactivated when the subroutine is entered. So, before a RETURN is made, the ON SQ(n) GOSUB m command must be issued again.

It's possible to use this Sound Queue Interrupt to control the flow of a program, but I can't really see any advantages of this. However, applications of the technique in machine code routines may prove useful. So, here are the details.

The techniques consist of setting up a sound of zero volume and tone, and putting it on 'Hold'. Releasing such a sound with an ON SQ(n)... interrupt active will result in the appropriate subroutine being entered. The below program demonstrates this:

```
100   FOR I=0 TO 3:GOSUB 220:NEXT I
110   count=0
120   ON SQ(1) GOSUB 18
121   :
130   a$=INKEY$
140   count=count+1
150   PRINT count
160   IF a$<>"" THEN RELEASE 1
170   GOTO 130
171   :
180   PEN 3:PRINT "Ouch!!":PEN 1
190   GOSUB 220
200   ON SQ(1) GOSUB 180
210   RETURN
211   :
220   SOUND 65,0,1,0
230   RETURN
```

Line 100 primes the Sound Queue, the SOUND statement in line 220 using Channel 1, tone 0, duration 1 and volume 0. The main loop of the program, from 130 to 170, simply increments and prints a variable, while checking for a keypress. Once one is detected, a RELEASE 1 command is issued and this causes the subroutine at 180 to be entered. In this, line 190 sets up the sound again and line 200 turns on the ON SQ() interrupt again. RUN it and press keys to see the routine work.

Envelopes

Not something for sticking letters in, but a means of altering either the pitch or amplitude (loudness) of a sound while that sound is being played. The amplitude is controlled by an AMPLITUDE ENVELOPE, set up by the ENV command, and the pitch by a TONE ENVELOPE, set up by the ENT command.

Let's now examine the actual business of designing Envelopes and also see some useful examples.

Amplitude Envelopes

The usual way in which Amplitude Envelopes are described is to use the ADSR Envelope. This stands for Attack-Decay-Sustain-Release. Figure 2.4 shows a typical ADSR Envelope.

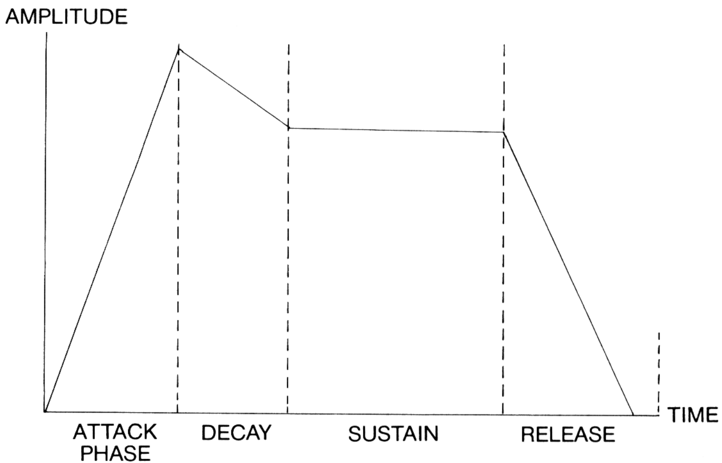


FIGURE 2.4

This is the type of Envelope that is possessed by the sounds produced by musical instruments. As you can see from Figure 4.4, such an Envelope is split into 4 sections, each of which we can define and put into an ENV statement, which can define an Envelope of up to 5 sections, each section requiring 3 parameters. An ENV command is of the form:

```
ENV n,henv,penv
or
ENV n,nstep,sizestep,ptime
```

'n' is the Envelope Number, 'henv' the Hardware Envelope Number and 'penv' the Envelope Period. We'll examine the Hardware Envelopes later in the Chapter when we look at directly accessing the Programmable Sound Generator from BASIC using the SOUND command we developed in Chapter 4. 'nstep' is the number of steps to be used in a software Envelope, 'sizestep' is the size of each such step in terms of volume and 'ptime' is the time between steps, in 1/100 seconds. More details will be found in the User Guide.Legal values of 'nstep' are between 0 and 127, values of 'sizestep' are between -128 and +127 and 'ptime' values between 0 and 255.

Simply issuing ENV n without any parameters will simply delete Amplitude Envelope 'n' from memory.

ENV

This simple program allows you to enter parameters for each of the four sections of an ADSR Envelope and see the Envelope displayed graphically, thus allowing you to see roughly what the Envelope will sound like. In addition, a list of ENV parameters is available and the effect of the Envelope can be heard.

Using ENV

Running the program will present you with a screen split into two parts, as shown in Figure 2.5. The scales are NOT shown on the screen.

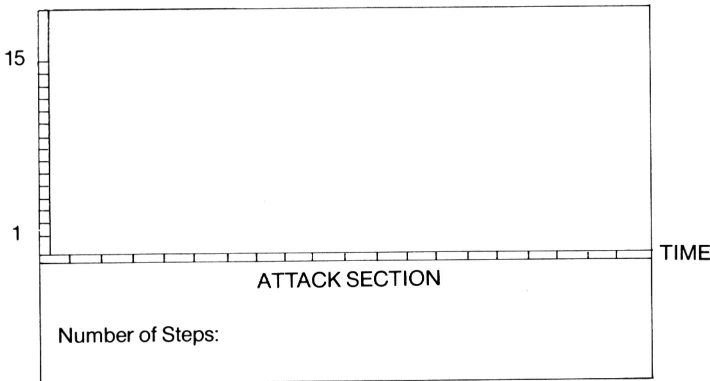


FIGURE 2.5

ENV screen.

Pressing ESCAPE twice will re-run the program. At each stage, the parameter of interest then press [ENTER]. Once a 'section' of an Envelope has been entered it will be drawn on the graph. To hear what you've produced at any time enter '#' instead of a number. This will play the sound and also display the ENV parameters. Once an Envelope is set up, pressing [ENTER] in response to a prompt without entering a number will lead to the current value of the parameter being preserved. This can also be used to set a parameter to zero if it hasn't been used before. Experiment with the program a little to get the hang of it. You could easily modify it to give a different time scale, say 0 to 3 seconds instead of 0 to 6 seconds by altering the value of the 'xs' variable. Note that if you design an Envelope with only two or three sections in it using ENV. you'll have to execute an ENV command containing only two sections to hear the sound properly; the sound produced in the program can occasionally be misleading if there aren't four sections to the Envelope.

It is possible to alter the program to accomodate Tone Envelopes. (See later). In this program, the 'Number of Steps' is the number of actual volume changes you want in this section of the Envelope. The 'Size of Step' is the amount by which the volume of the sound changes with each step, and the last parameter is the duration of each step in units of 0.01 seconds. Note that values for the 'Size of Step' accepted by thie program are limited to the range -15 to +15. Once an Envelope is developed it can be put into a program of your own. An important point to remember is that the Envelope must be defined before you try and use it with a SOUND command.

```
100 REM ENV program
110 REM Joe Pritchard
120 REM October 1985
125 ON BREAK GOSUB 10000
130 :
140 :
150 :
160 MODE 1
170 GOSUB 290
180 GOSUB 450
190 ORIGIN 10,120,10,640,400,120:CLG:MOVE 0,0,2:FOR i=1
    TO section:DRAWR x%(i),y%(i):NEXT
200 FOR section=1 TO 4
210 GOSUB 580
220 GOSUB 720
230 NEXT section
```

```
240 PAPER#2,3:PEN#2,1:CLS#2:PRINT#2:PRINT#2:PRINT#2,"
    Press a key to go on":PAPER#2,0:CALL &BB18:CLS#2
250 GOTO 190
260 :
270 :
280 :
290 REM initialise routine
300 sf%=0
310 ys=17:xs=1
320 DIM name$(4)
330 DIM numstep%(4)
340 DIM stepsize%(4)
350 DIM ptime%(4)
360 DIM x%(5)
370 DIM y%(5)
380 RESTORE 410
390 FOR i%=1 TO 4:READ name$(i%):NEXT
400 WINDOW #2,1,40,20,25
410 DATA "Attack ","Decay  ","Sustain","Release"
420 RETURN
430 :
440 :
450 REM draw axes
460 ORIGIN 0,100,0,640,400,100
470 CLG:MOVE 0,0,1
480 MOVE 0,0:DRAW 650,0
490 MOVE 0,10:DRAW 650,10
500 MOVE 0,0:DRAW 0,400
510 FOR y=30 TO 275 STEP 17
520 MOVE 0,y+8:DRAW 5,y+8
530 NEXT
540 FOR x=10 TO 640 STEP 50:MOVE x,0:DRAW x,5:NEXT
550 RETURN
560 :
570 :
580 REM enter an envelope section
590  PAPER#2,3:PEN#2,1:LOCATE#2,10,1:PRINT#2,name
    (section Section."
600 n$="Number of Steps":sf%=0:lower%=0:upper%=127:GOSUB
900
610 IF sf%=1 THEN GOTO 600
620 IF no%=0 THEN numstep%(section)=num%
630 n$="Size of Steps":lower%=-15:upper%=15:sf%=0:GOSUB 900
```

```
640 IF sf%=1 THEN GOTO 630
650 IF no%=0 THEN stepsize%(section)=num%
660 n$="Pause Time":sf%=0:lower%=0:upper%=256:GOSUB 900
670 IF sf%=1 THEN GOTO 660
680 IF no%=0 THEN ptime%(section)=num%
690 RETURN
700 :
710 :
720 REM draw envelope
730 ORIGIN 10,120,10,640,400,120:CLG:MOVE 0,0,2
740 pt=ptime%(section):IF pt=0 THEN pt=256
750 x%(section)=numstep%(section)*pt*xs
760 y%(section)=ys*stepsize%(section)*numstep%(section)
770 FOR i=1 TO section:DRAWR x%(i),y%(i):NEXT
780 RETURN
790 :
800 :
810 REM check a string as a number
820 IF a$="" THEN no%=1:RETURN ELSE no%=0
830 IF a$="#" THEN GOSUB 970:RETURN
840 IF (LEFT$(a$,1)<"0" OR LEFT$(a$,1)>"9") AND LEFT$(a$,1)<>"-"
    THEN f%=1 ELSE f%=0
850 IF f%=1 THEN SOUND 1,100:RETURN
860 num%=VAL(a$)
870 RETURN
880 :
890 :
900 REM enter a parameter
910 PAPER #2,0:PEN#2,3
920 LOCATE#2,1,3:PRINT#2,n$+":
    ":LOCATE#2,1,3:PRINT#2,n$+":
    ";:PEN#2,1:INPUT#2,"",a$:GOSUB 810:IF no%=1 OR sf%
    =1 THEN RETURN
930 IF f%=1 OR (num%<lower% OR num%>upper%) THEN GOTO
910
940 RETURN
950 :
960 :
970 REM sound routine
980 ENV 1,numstep%(1),stepsize%(1),ptime%(1),
     numstep%(2),stepsize%(2),ptime%(2),numstep%(3),
     stepsize%(3),ptime%(3),numstep%(4) ,stepsize%(4),ptime%(4)
```

```
 990 SOUND 1,200,0,0,1
1000 PAPER #2,3:PEN#2,1:CLS#2:PRINT#2,"ENV
     1,";numstep%(1)",",stepsize%(1)",",ptime%(1)",",numstep%
     (2)",",stepsize%(2)",",ptime%(2)",",num
     step%(3)",",stepsize%(3)",",ptime%(3)",",numstep%
     (4)",",stepsize%(4)",",ptime%(4)
1010 PAPER #2,1:PEN #2,3:LOCATE #2,6,4:PRINT#2,"Type F to
     Finish, C to go on"
1020 a$=INKEY$:IF a$="" THEN GOTO 1020
1030 IF INSTR("FfCc",a$)=0 THEN SOUND 1,100:GOTO 1020
1040 IF INSTR("FfCc",a$)<3 THEN ON BREAK STOP:END
1050 PAPER
     #2,0:CLS#2:PAPER#2,3:PEN#2,1:LOCATE#2,10,1:
     PRINT#2,names
     (section);" Section.":PAPER#2,0:PEN#2,1:
     a$="":sf%=1:RETURN 10000 RUN
```

ENT

The ENT command has the syntax:

```
ENT n,tonep,ptime
or
ENT n,stepnumber,stepsize,ptime
```

n is the Tone Envelope Number, and 'ptime' and 'stepnumber' are the same as for ENV. However, the maximum number of steps is now 239. For ENT, however, 'stepsize' is a parameter in the range -128 to +127 that controls the way in which the pitch of the sound changes; a negative value causes a decrease in pitch and a positive value causes an increase in pitch.

For the two parameter ENT command, 'tonep' is a new value for the pitch of a sound played with this Envelope.

Altering ENV to allow Tone Envelopes

The process of changing the program is fairly straight forward. Simply alter the program by adding or altering the following lines.

```
190  ORIGIN 10,120,10,640,400,120: CLG:MOVE 0,100,2:
     FOR I=1 TO section: DRAWR  x%(i),y%(i):NEXT

310  ys=0.1:xs=1

510  FOR y=30 TO 275 STEP 20
```

```
600  n$="Number of Steps": sf%=0: lower%=0:
     upper%=239: GOSUB 900

630  n$="Size of Steps": lower%=-128: upper%=+127:
     sf%=0: GOSUB 900

730  ORIGIN 10,120,10,640,400,120: CLG: MOVE 0,100,2

980  ENT 1, numstep%(1), stepsize%(1), ptime%(1),
     numstep%(2), stepsize%(2), ptime%(2), numstep%(3),
     stepsize%(3), ptime%(3), numstep%(4), stepsize%(4),
     ptime%(4)

990  SOUND 1,1000,0,15,0,1

1000 PAPER#2,1: PEN#2,3: LOCATE#2,6,4: PRINT "ENT
     1,";numstep%(1)", "stepsize%(1)"," ptime%(1)","
     numstep%(2)"," stepsize%(2)"," ptime%(2)","
     numstep%(3)"," stepsize%(3)"," ptime%(3)","
     numstep%(4)"," stepsize%(4)"," ptime%(4)
```

The scaling of the axes is also different; horizontally the time scale is the same, but vertically the lowest mark is '200', and after that each marker is 200 period units apart. The ENT program starts it s sounds with a period of 1000 as it's tone parameter rather than zero, and it is important to remember that with decreasing values of this parameter lead to sounds of an increasing pitch.

Designing Envelopes

We'll now look at the business of designing some simple Envelopes; but first, the problems that can be encountered.

One phenomenon to look out for when you are designing Amplitude or Tone Envelopes is that of 'Cycling'. For example, exceeding a volume of 15 will result in an actual volume of:

```
volume MOD 16
```

Thus a sound can go from being very loud (15) to inaudible (16) in one step. This can be useful, but it is useful to be aware of this fact. It is commonly caused by setting up an envelope that takes the value of the amplitude above 15 or below 0. A similar thing can happen with Tone Envelopes if we've set up an Envelope that takes the tone

generated below zero. This produces a 'buzzing' sound. The practical difficulty that this problem presents is that Envelopes are not universally applicable to sounds of all amplitudes and frequencies.

Stepping

The Envelopes set up are not 'continuous', but alter pitch or amplitude in a stepwise fashion. This is particularly so with the Amplitude Envelope, where there are relatively few steps available. The fact that this is a limitation of the Programmable Sound Generator and not of the Firmware means that it is almost impossible to ensure a smooth change in volume. Smaller steps can be accommodated by the Tone Envelope, but it is still possible to hear the transition from one tone level to another as a sudden 'step' rather than as a smooth change.

The graphs drawn by the ENV program, are, therefore, not truly accurate representations of how the Amplitude and Tone will change with time.

Waveform

The PSG in the Amstrad machines produces a square wave. Musical instruments produce a complex waveform based on the sine wave. If you want to see the difference, look at Figure 2.6. Even after we add our Tone and Amplitude Envelopes, we're still operating on a square wave. For this reason, a computer generated 'piano' sound, for example, will not sound the same as a real one. The waveform of a sound gives it 'tonal quality', the property that makes a note of the same frequence played on, say, a flute and an oboe different. You are effectively stuck with the square wave tonal quality; you could try filtering the output from the Amstrad and passing it to an external amplifier, but this would have little effect and would not be reproducible on other Amstrads. One point about the square wave is that it is very rich in harmonics.

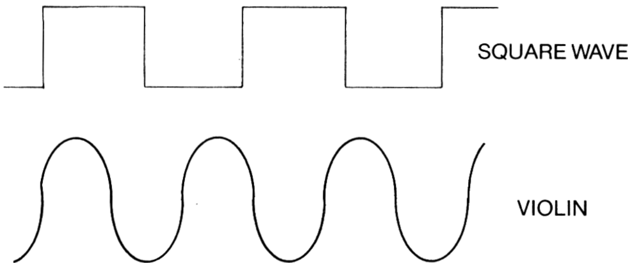


FIGURE 2.6

One approach that might be useful is to play tones at the same time on different channels; this appears to produce some alternation in the sound of the tone played. However, much of sound generation is subjective, and so you must experiment to see if you can detect any difference.

Designing Amplitude Envelopes

Before we go any further, it's useful to remember that you can also vary the amplitude of a sound by using FOR...NEXT loops to alter SOUND command parameters. Of course, this requires the total attention of the computer, but is occasionally useful. The technique can also be used in conjunction with ENV as we'll soon see. If ENV is used to produce an Envelope with fewer than 4 sections in it, the ENV command should only contain the sections actually used.

Modulation

This is the technical term for the process of continuously altering the volume of a sound while it's being played. This is probably the simplest amplitude Envelope available to us, and can be done with the below ENV command. Type the parameters into the ENV program to see what it looks like.

```
ENV 1, 5, 1, 2, 5, -1, 2, 5, 1, 2, 5, -1, 2
```

This takes the amplitude up by 5 units in 0.1 seconds, then down by 5, then repeats. The command:

```
SOUND 1,200,-32767,0,1
```

will repeat this envelope 32767 times, taking the volume from 0 to 5 and back again.

```
SOUND 1,200,-32767,5
```

will take the volume from 5 to 10 and back again. While the sound is being produced, alter the envelope to:

```
ENV 1, 5, 1, 2, 5, -2, 2, 5, 1, 1
```

This will result in the sound changing as the Envelope changes, and now the sound starts at 5, goes up to 10, then down to 0, then back up to 5. To see this type of Amplitude Envelope, where the amplitude starts at some value other than zero, make the following alterations to ENV.

```
190    ORIGIN 10,120,10,640,400,120: CLG:MOVE 0,5*ys,2:
       FOR I=1 TO section: DRAWR x%(I),y%(I):NEXT

730    ORIGIN 10,120,10,640,400,120: CLG:MOVE 0,5*ys,2

990    SOUND 1,200,0,5,1
```

This allows the Envelopes to be displayed with a starting volume of 5. Again, remember that playing this three section Envelope from within the ENV program, which expects a 4 section Envelope, may produce odd sounding results. Such an Envelope can also, of course, be applied to noise as well as tone.

Echo

It's useful to have an 'Echo' type sound effect to apply to sounds, and this usually requires that we use a combination of Envelopes and a FOR...NEXT loop to get a satisfactory sound. Try the below routine to hear a reasonable 'echo'.

```
10 FOR I=15 TO 5 STEP -5
20 ENV 1,1,0,1,5,-(I/5),10
30 SOUND 1,200,-1,I,1
40 NEXT
```

Note how we use '-1' in line 30 to ensure that the Envelope is played once. In addition, we've created a 'dynamic' envelope here, in which the size of the step in the second section of the Envelope depends upon the start value of the volume. This is necessary to prevent cycling problems. The Envelope consisted of a start at a given volume, then a decay to zero. This can form the basis of other effects, especially when noise is used. For example, for an odd noise that is reminiscent of a cat's purring, replace line 30 above with:

```
30 SOUND 1,2000,-1,I,1,0,15
```

A similar Envelope can be used for a 'Snare Drum' effect:

```
20 ENV 1,1,0,1,5,-(I/5),6
30 SOUND 1,299,-1,I,1,0,5
```

A final effect using an Envelope of this type is a 'gunshot' or explosion.

```
10 ENV 1,3,5,1,1,0,3,5,-2,10,5,-1,10
20 SOUND 1,1,0,0,1,0,31
```

Musical Instruments

When trying to simulate musical instruments with the PSG we always need an Envelope of some sort. In this section I want to look at the general procedure for designing such Envelopes. First of all, you have to know roughly what sort of ADSR envelope an instrument produces, AND have a knowledge of the HARMONICS produced by the instrument. First of all, what's a Harmonic?

Well, when a tone is generated, say of FREQUENCY n, an instrument will also generate frequencies of 2*n, 3*n, 4*n and so on. The magnitude of these Harmonics with respect to each other, combined with the Envelope used, gives the musical instrument it s characteristic sound. To hear a tone and it s harmonics, RUN the below program.

```
10 PRINT "This is a low 'C'"
20 SOUND 1,956,300,15
30 PRINT:PRINT:PRINT
40 PRINT "When it's finished"
50 PRINT "Press any key."
60 CALL &BB18
70 SOUND 65,956,300,15
80 SOUND 66.478.300.15
90 SOUND 68,239,300,15
95 RELEASE 7
```

Channel A plays the FUNDAMENTAL, or note of interest, Channel B the second harmonic and Channel C the third harmonic. The 'tone' parameter for the second and third harmonic can be obtained by:

second=fundamental 'tone' parameter/2
third=fundamental 'tone' parameter/3

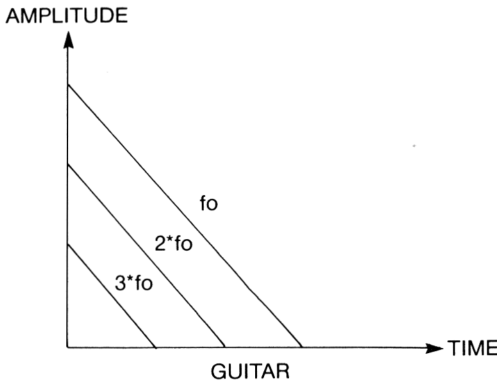
This can cause problems when we're playing with Programmable Sound Generators; the registers of the PSG, which are programmed by the SOUND command, can only hold integer values. If we have a fundamental of, say, 239, then 239/2 gives a non integer result, which cannot be represented accurately in the PSG registers. This will give 2nd and 3rd harmonics that are not quite right.

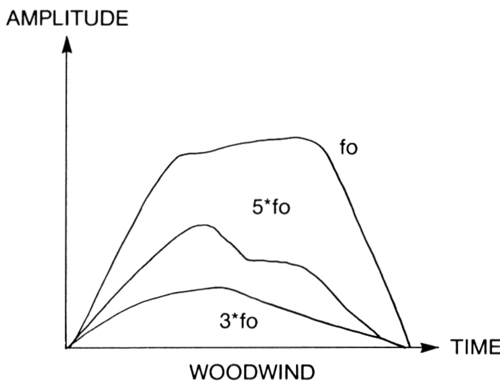
The volume of harmonics often decrease as their order (2nd, 3rd, 4th..etc.) increases, but some instruments still produce sounds that have a strong contribution from 7th or 8th harmonics. This is obviously a problem for us on our three channel sound generator! So, we'll be limiting ourselves to a maximum of the fundamental and two

harmonics, the actual harmonics used depending upon the instrument we're trying to simulate.

A further problem with the harmonics generated by some instruments is that the exact proportions of harmonics in a sound can vary with the actual pitch of the note played. This means that for some instruments, we could get a simulation of the sound produced by the instrument only over a certain range of sounds.

We can easily get our 'mix' of harmonics by simply altering the volume of each Channel. Once we've got our harmonic mix, we apply an Envelope to it, and that's it. Figure 2.7 shows Envelopes for each of the 3 most significant harmonics for a few instruments.





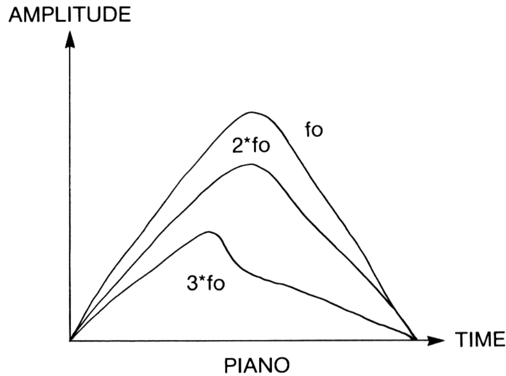


FIGURE 2.7

As can be seen, the ADSR Envelope applied to each harmonic can also be different. The listing below is a simulation of a woodwind instrument.

```
10 ENV 1,1,15,6,1,-1,16,7,-2,2
20 ENV 2,1,9,6,1,-1,16,3,-2,1,2,-1,5
30 ENV 3,1,6,6,1,-1,16,2,-1,3,1,-1,6
40 SOUND 65,750,-1,0,1
60 SOUND 68,250,-1,0,3
70 RELEASE 7
```

A couple of points about this: firstly, we use RELEASE to synchronise the three sounds. Secondly, each envelope should take about the same amount of time to execute. An interesting point here is that the fifth harmonic is 'louder' than the third harmonic. Finally, some harmonics are always produced when we generate square waves like this; we're simply controlling the proportions here.

You might like to try setting up Envelopes for other instruments using the principles outlined here. In addition, noise could be added; this will give a further quality to the sound generated.

Tone Envelopes

The simplest Tone Envelope to apply to a tone is often called 'Vibrato', and is a fluctuation in the pitch of the tone. Try the below program to hear this effect in action.

```
10 ENT -1,10,2,1,20,-2,1,10,2,1
20 SOUND 1,200,300,10,,0
30 SOUND 1,200,300,10,,1
```

The only thing to note about this program is the use of the negative Envelope number in line 10. This ensures that when the tone envelope is applied to a sound the envelope will be repeated for as long as the sound is playing.

Glissando

In this effect we get a constant fall or rise in frequency, a little like a falling bomb, or a trombone sound.

```
10 ENT -1,100,-2,1
20 SOUND 1,500,250,,,1
```

This effect can be applied to musical notes by gradually changing the pitch of one note to that of the next note to be played. In this sort of application, it's probably best to use FOR...NEXT loops to change the pitch rather than a tone envelope. Try:

```
10 SOUND 1,200,100
20 FOR period=200 TO 300
30 SOUND 1,period,1
40 NEXT period
50 SOUND 1,300,100
```

The Programmable Sound Generator

The Amstrad computers use the AY-3-8912 Programmable Sound Generator, or PSG. We are usually insulated from it by the SOUND, ENV and ENT commands, although machine code programmers have to access the PSG directly.

The PSG is buried deeply in the hardware and isn't easily accessed via the IN and OUT instructions from BASIC. Apart from the lack of accessibility, it's generally ill advised to try and access the PSG through any other route than the Firmware MC SOUND REGISTER routine. This routine ensures that the data is sent to the PSG registers only when the act of writing to the PSG will not affect anything else that the PPI is doing. This is how the |SOUND command that we add in Chapter 4 works. We will be using this command to write data to the PSG registers.

The 8912, as it's known to it s friends, is a three channel device, capable of producing both tone and noise. Amplitude Envelopes are

also provided by the chip, but they are 'built in' to the device and we cannot change any of them. Tone Envelopes are not provided. In addition, it has an Input/Output port, which is used in the Amstrad for keyboard/joystick control. Setting up a sound directly to the PSG often involves writing to several registers, and you should remember that when you're doing this you are bypassing all the Sound Queues, etc. offered by the Sound Manager of the Firmware.

The PSG has 15 registers, numbered 0 to 14; if you should consult the General Instruments data sheet for this device you may well see reference to Register 15. Don't panic; there exists a device called the AY-3-8910 which has an extra Input/Output register, Register 15. Simply ignore any such reference to Register 15 if you've got on 8912.

The PSG Registers

Register 0 and 1.

These control the pitch of the tone to be played on Channel A. Together they form a twelve bit register, with the lower 8 bits being held in Register 0 and the upper 4 bits in Register 1. The value held in these two registers thus defines the pitch of the tone generated. The higher the value in this register, the lower the tone generated. The value to put in this twelve bit register to get a particular frequency is given by:

value=125000/frequency

Thus to write the value 255 to the Channel A tone control register, we'd issue the commands,

```
SOUND 0,255: SOUND 1,0
```

assuming, of course, that we've installed the RSX commands! Register 0, by the way, is called the FINE TUNE register, and Register 1 is called the COARSE TUNE register.

Registers 2 and 3.

These perform the same function as Registers 0 and 1 but for Channel B.

Registers 4 and 5.

These perform the same function as Registers 0 and 1 but for Channel C.

Register 6.

This is the Noise Pitch Control register, and is a 5 bit register. A value of 31 will give the lowest noise frequency available, and one of 0 will give the highest noise pitch available.

Register 8,9 and 10.

These are the Amplitude Control Registers for the three Channels. Register 8 controls the volume of Channel A, 9 that of B and 10 that of C. All the registers are effectively 5 bit registers. Values between 0 and 15 set the volume of a Channel in the same way as the volume parameter of the normal SOUND command. That is, 0 gives silence and 15 gives maximum volume. If bit 4 of the register is set to '1', corresponding to a value in the register of 16 or above, then the status of bits 0 to 3 is ignored and sound on the Channel is played under the control of an Amplitude Envelope. More on this later.

Even after you've set up the Tone and Amplitude registers for a given Channel, you still won't hear much. The reason for this is that for sound to be played on a Channel that Channel must be ENABLED. This is controlled by Register 7.

Register 7

This is a control register, arranged as a series of bits, each bit controlling one aspect of the behaviour of the PSG.

Figure 2.8 shows the layout of this register.

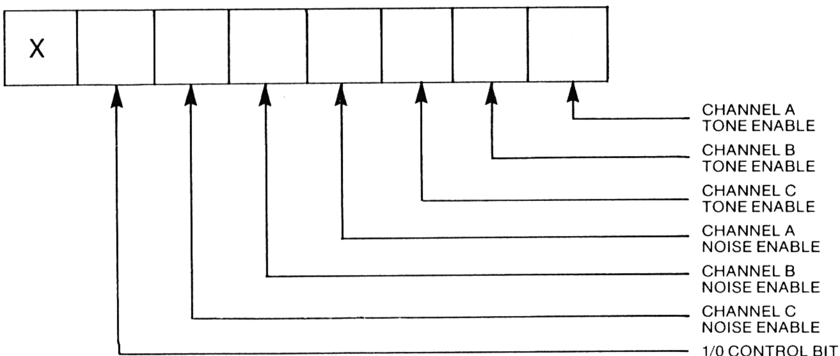


FIGURE 2.8

The operation is quite simple; setting bit 0 to '0' will allow a tone of the pitch specified by Registers 0 and 1, and of the Amplitude set by Register 8, to be played on Channel A. These bits are all 'active low' bits; for tone or noise to be played on a channel the relevant bits must be set to '0' rather than '1'. Setting a bit to '1' disables tone or noise on the appropriate Channel.

Bit 6 controls whether the Input/Output port is to be used for output or input. The Amstrad expects this port to be configured for Input, and so this bit should be set to '0'. Bit 7 isn't used in this version of the PSG.

This register is at the heart of PSG operations; for example, we can set up tones that are 'held' and then play them by 'releasing' them by simply setting the appropriate bits of Register 7 to '1', setting up the Amplitude and Tone Control Register for each Channel and then setting the appropriate bits of R7 to '0' to play the notes simultaneously. For example, the below commands will play tones on both Channels A and B.

```
10 SOUND,7,&X00111111
20 SOUND,0,255
30 SOUND,1,1
40 SOUND,2,257
50 SOUND,3,1
60 SOUND,8,10
70 SOUND,9,10
80 SOUND,7,&X00111100
```

Line 10 turns off both noise and tone on all channels. Lines 20-30 set up a tone on Channel A, lines 40-50 set up a tone on Channel B and lines 60-70 set up the Amplitude Control Registers. Finalkly, line 80 enables tone on Channel A and B. To mix some noise in to Channel A, replace line 80 with:

```
80 SOUND,7,&X00110100
```

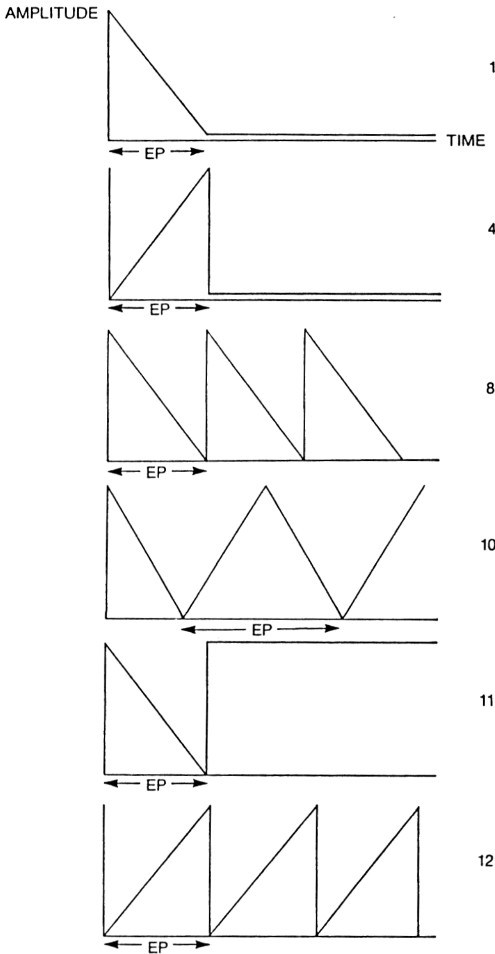
If you run this program you will hear the sound which will go on....and on...and on! There is no duration register in the PSG, and we have to deliberately turn the appropriate Channels off by either setting the Amplitude to 0 or by disabling the Channel by setting the appropriate bit in Register 7.

We now come to the Amplitude Envelopes that the PSG supports. There are 8 of these, and the one selected depends upon the value in Register 13. This is called the Envelope Shape Register. Figure 2.9

shows the different Envelopes available and the values of R13 needed to produce them. Other values in this register will simply reproduce a couple of these Envelopes.

All of the volume Envelopes have a maximum volume of 15 and a minimum of 0. Remember that to set a Channel to play a sound under an Envelope, bit 4 of that Channel's amplitude control register must be set to '1'.

In Figure 2.9, EP stands for Envelope Period, which is the duration of the Envelope that is being played.



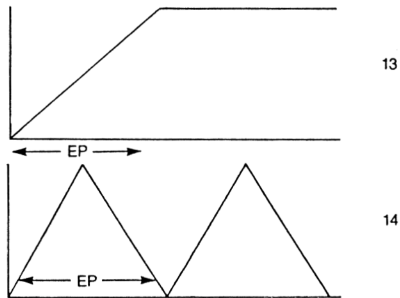


FIGURE 2.9

The duration of the Envelope is set by the Envelope Period Registers, Register 11 and Register 12. R11 holds the least significant 8 bits and Register 12 the most significant 8 bits of the value. These two registers hold the duration of the Envelope in milliseconds. To hear a PSG Envelope in action, try the below program:

```
10 SOUND,7,&X00111111
20 SOUND,0,255
30 SOUND,1,1
40 SOUND,8,16
50 SOUND,11,0
60 SOUND,12,4
70 SOUND,7,&X00111110
```

Again, this will go on until it's stopped. A quick way of doing this in direct mode is to press the CLR key.

These Envelopes are the ones set up by the two parameter version of the ENV command. They are very useful, and can be applied to either tone or noise. However, we cannot alter them in any way.

There are two other RSX commands that we add in Chapter 4 of interest here. These allow us to get the parameters of an Envelope back from memory so that we can inspect it. These commands are TONE and |AMPLITUDE, and their use is described in Chapter 4.

To finish this Chapter, two useful subroutines. The first of these I call 'PLAY', which simply takes a string of characters representing musical notes and 'plays' them on a single channel. It is based on a command present in MSX BASIC, although the subroutine here is no where near as complicated.

The listing is given below and contains a short demo. The notes are passed over to the subroutine in 'm$'. The routine is straight forward, and you will no doubt be able to add more 'commands' to the subroutine. The only points to note are that the RESTORE statements are to specific lines in the program, and if you alter the positions of these DATA statements, which contain the pitch information for the notes in each of the three octaves that the subroutine handles, you must alter the RESTORE commands accordingly. In addition, if the subroutine is used in your own programs, the parameters such as 'octave', 'tempo' , 'duration%' and 'dot', as well as the array 'note%' must be initialised before use.

Notes

Notes are represented in the string by the Upper Case letters for that note. Where sharps are available, they are represented by the Lower case letter for that note. If you attempt to input letters that are not Notes or Commands, then the ERROR command is used to generate an error.

('Improper Argument'). In the above program the default Octave is set to 2, and so the notes played will be from this Octave, which is the one containing middle 'C'.

```
20000 REM PLAY subroutine and demo
20010 REM Joe Pritchard August 1985
20020 :
20030 :
20040 DIM note%(12)
20050 st$="CcDdEFfGgAaBLORT.1234567890"
20060 tempo=1
20070 duration%=32
20080 octave=2
20090 dot=1
20100 :
20110 REM these parameters are set up
20120 REM just the once
20130 :
20140 :
20150 m$="T0.25C....cDdR20EFO1fGgL64AaB"
20160 GOSUB 20210
20170 END
20180 :
20190 :
20200 :
```

```
20210 REM PLAY subroutine
20220 GOSUB 20460
20230 :
20240 REM the above sets up the notes for
20250 REM the octave of interest
20260 :
20270 FOR i%=1 TO LEN(m$)
20280 a$=MID$(m$,i%,1)
20290 opt%=INSTR(st$,a$)
20300 IF opt%=0 THEN ERROR 5
20310 IF opt%<13 THEN GOSUB 20380:GOTO 20340
20320 opt%=opt%-12
20330 ON opt% GOSUB 20600,20660,20740,20810,20870
20340 NEXT i%
20350 RETURN
20360 :
20370 :
20380 REM play a note
20390 a$=MID$(m$,i%+1,1)
20400 IF a$="." THEN i%=i%+1:dot=1.5 ELSE dot=1
20410 SOUND 1,note%(opt%),tempo*duration%*dot,15
20420 RETURN
20430 :
20440 :
20450 REM once an octave is read in, set it
20460 IF octave=1 THEN RESTORE 20920
20470 IF octave=2 THEN RESTORE 20930
20480 IF octave=3 THEN RESTORE 20940
20490 FOR j%=1 TO 12:READ note%(j%):NEXT
20500 RETURN
20510 :
20520 :
20530 REM read a number from string
20540 num$="":i%=i%+1
20550 IF i%>LEN(m$) THEN RETURN
20560 a$=MID$(m$,i%,1):IF (a$<="9" AND a$>="0") OR a$="." THEN
        num$=num$+a$:i%=i%+1:GOTO 20550
20570 i%=i%-1:RETURN
20580 :
20590 :
20600 REM Ln is the duration of notes
20610 GOSUB 20530
20620 duration%=VAL(num$)
```

```
20630 RETURN
20640 :
20650 :
20660 REM get an octave command
20670 GOSUB 20530
20680 octave=VAL(num$)
20690 IF octave<1 OR octave>3 THEN ERROR 5
20700 GOSUB 20460
20710 RETURN
20720 :
20730 :
20740 REM rest routine
20750 GOSUB 20530
20760 rest%=VAL(num$)
20770 SOUND 1,0,rest%*tempo*dot,0
20780 RETURN
20790 :
20800 :
20810 REM Tempo routine
20820 GOSUB 20530
20830 tempo=VAL(num$)
20840 RETURN
20850 :
20860 :
20870 REM dot routine
20880 REM just in case one gets through!!
20890 RETURN
20900 :
20910 :
20920 DATA 239,225,213,201,190,179,169,159,150,142,134,127
20930 DATA 119,113,106,100,95,89,84,80,75,71,67,63
20940 DATA 60,56,53,50,47,45,42,40,38,36,34,32
```

In addition to us simply putting notes in the string, there are also a few single letter 'commands' which allow you to alter the way in which the notes are played.

O. The Octave command, followed by a single digit parameter, sets the octave that is to be used by subsequent notes. Legal values are 1,2 and 3, 1 selecting the Octave below Middle 'C', 2 the Middle 'C' Octave and 3 the Octave above Middle 'C'. No other parameter values are legal. If you use PLAY in your own programs, set up this

parameter before calling the PLAY subroutine. An example of its use is:

m$="O3Cc"

T. The Tempo command sets the 'speed' at which the notes are played, and has a single parameter. The lower the value of the parameter (non integer values allowed), the faster the notes are played. For example:

m$="T3"

L. This letter, followed by an integer number, sets the duration of all subsequent notes. I would have used 'D' for this command, but we've already got a note 'D'.....

R. The Rest command takes a single integer parameter, and generates a pause of that for that many note durations. As with note lengths, the length of the rest depends upon the Rest parameter and the Tempo.

The '.' simply extends the length of the preceding note by 1.5 times. Only one dot is allowed after a note.

RHYTHM

This subroutine is similar to PLAY but allows you to put together simple rhythms consisting of a drum, snare and 'poom!' sounds beloved of all owners of drum synths! (Although, of course, ours aren't so good . . .)

The principles are the same as those behind the PLAY routine. A string of characters are passed to the subroutine in 'm$'. The 'commands' available are as follows.

D. This simulates a 'drumbeat' sound.

S. This gives a Snare Drum sound.

P. This provides a 'Poom!!" sound.

L. As with PLAY, this sets the length of drumbeats. 'L' is followed by a single integer parameter.

T. This sets the Tempo and is followed by a numeric parameter that can be fractional if necessary. However, some fractional values, in

conjunction with some values for the 'L' parameter, will result in very long sounds being produced.

You could easily combine these two routines into one program by having 'PLAY' use one Channel and 'RHYTHM' use a second Channel.

```
20000 REM RHYTHM subroutine and demo
20010 REM Joe Pritchard August 1985
20020 :
20030 :
20050 st$="DSPLRT.1234567890"
20060 tempo=1
20070 duration%=8
20095 GOSUB 20920
20100 :
20130 :
20140 :
20150 m$="T0.125L4DDR160L8SL4DDR160L8SSR200L8PL4R200"
20160 GOSUB 20210
20170 GOTO 20160
20180 :
20190 :
20200 :
20210 REM PLAY subroutine
20270 FOR i%=1 TO LEN(m$)
20280 a$=MID$(m$,i%,1)
20290 opt%=INSTR(st$,a$)
20300 IF opt%=0 THEN ERROR 5
20310 IF opt%<4 THEN GOSUB 20380:GOTO 20340
20320 opt%=opt%-3
20330 ON opt% GOSUB 20600,20740,20810,20870
20340 NEXT i%
20350 RETURN
20360 :
20370 :
20380 REM play a note
20410 GOSUB 20920
20420 IF a$="D" THEN SOUND 1,800,-1,0,1,0,31
20430 IF a$="S" THEN SOUND 1,0,-1,0,1,0,2
20440 IF a$="P" THEN SOUND 1,70,-1,0,1,1
20450 RETURN
20510 :
20520 :
```

```
20530 REM read a number from string
20540 num$="":i%=i%+1
20550 IF i%>LEN(m$) THEN RETURN
20560 a$=MID$(m$,i%,1):IF (a$<="9" AND a$>="0") OR a$="." THEN
        num$=num$+a$:i%=i%+1:GOTO 20550
20570 i%=i%-1:RETURN
20580 :
20590 :
20600 REM Ln is the duration of notes
20610 GOSUB 20530
20620 duration%=VAL(num$)
20625 GOSUB 20920
20630 RETURN
20640 :
20650 :
20720 :
20730 :
20740 REM rest routine
20750 GOSUB 20530
20751 d%=VAL(num$)
20770 GOSUB 20920:SOUND 1,0,tempo*d%,0
20780 RETURN
20790 :
20800 :
20810 REM Tempo routine
20820 GOSUB 20530
20830 tempo=VAL(num$)
20835 GOSUB 20920
20840 RETURN
20850 :
20860 :
20870 REM dot routine
20880 REM just in case one gets through!!
20890 RETURN
20900 :
20910 :
20920 REM set up the envelopes needed
20930 ENV 1,1,15,1,15,-1,duration%*tempo
20940 ENT 1,65,1,duration%*tempo
20950 RETURN
```

That completes our examination of the Amstrad Sound facilities. Sound is one area where experimentation is valuable, and I hope that this Chapter has given you some new ideas.

Chapter 3
Getting down to BASICs

A programmer can quite happily live his life and program his computer without ever knowing how his programs and their associated variables are stored. But, you can get a lot more from your machine if you have a knowledge of the way in which the programs are stored in the machine. So, in this Chapter we'll look at this aspect of programming the Amstrad.

BASIC programs are stored in the machine from address &0170 (368) upwards. The work in this book has been done on the Amstrad 464, but it is applicable to any of the other machines. For the other machines, it might be useful to confirm that program storage begins at this address. This can be done by the following steps.

1. Turn on the computer
2. i=0 (ENTER)
3. PRINT (@i-6) (ENTER)

This will return the address of the start of BASIC text storage space.

BASIC Program Storage

The arrangement of bytes that make up a program line is as follows:

low
high line length

low
high line number

text of the
program line

0 terminator

Line Length. This is the number of bytes in the program line, including the bytes representing line length, line number, and terminator.

Line Number. This is the number of the line, stored low byte first.

Program Text. This is the actual body of the line; we'll take a look at how different program structures are stored shortly.

Terminator. This zero byte is used to mark the end of a BASIC line.

A program consists, therefore, of a collection of these lines. The end of the program is marked by a 'line' consisting of line length and line number bytes, all set to zero.

Storage of statements

As you are probably aware, BASIC commands such as PRINT or REM are stored as single bytes in program lines. These single bytes are called TOKENS. Appendix 1 shows the Tokens used by the Amstrad. In addition to these, certain numbers are used by the Amstrad to indicate that a Real Variable is about to be encountered, or an Integer Constant, and so on. We'll soon see these in use.

Tokenisation occurs between you typing in the code and the representation being included in the BASIC program area. The process also marks occurrences of variables and constants, etc. We'll now look at how various commands are stored in the BASIC program area.

Statement Separation

The ':' is used to separate BASIC statements that occur on the same line; this is represented in tokenised form as the value 1.

Variable Assignments

Let's examine a typical line, such as:

```
1 y=n
```

where n is a numeric constant. A simple program can be used to PEEK out the contents of RAM, remembering that the program starts at

address 368. In all cases, the variable name will have its last character modified by having 128 added to its ASCII code. For single character variable names, that single character will have 128 added. In addition, the variable name will be prefixed by bytes indicating what type of assignment it is. The bytes used here vary for each variable type, and also differ depending upon how the variable type has been selected; in this discussion I'm assuming that the variable type has been selected by a type definition character after the variable name, rather than by the DEFINT, etc. commands.

Where 'n' is Integer

How 'n' is represented in the line depends on its size; a general description of the tokenised line 1 above is:

lenlow,lenhigh, 1,0, t1,t2,t3, 249, 239, number, 0

Here, t1, appears to be the type identifier. The values of t2 and t3 vary in different assignments, and those given here will be for illustrative purposes only. I have not worked out their precise meaning. 249 is (ASC("y")+128) and 239 is the token for '='. The number of bytes used to represent 'number' varies, and so will the line length. In each of the following tables, remember that t2 and t3 values are illustrative ones only.

'n' between 0 and 9 inclusive

t1	13
t2	5
t3	0
'number'	('n'+14)

Thus 5 is stored with a single 'number' byte set to 19.

'n' between 10 and 255 inclusive

t1	13
t2	5
t3	0
'number'	25, value of n

Here 'number' is a two byte value.

'n' between 256 and 32767 inclusive

t1	13
t2	5
t3	0
'number'	26,nlow,nhigh

Here 'number' is three bytes long, with the two bytes representing 'n' in the usual low-high order.

'n' greater than 32767

t1	13
t2	5
t3	0
'number'	31,n1,n2,n3,n4,n5

Here, 'number' is stored as a Floating Point number. The format of these numbers will be looked at later in the Chapter. 31 indicates to the BASIC Interpreter that a Floating Point Constant is coming.

With negative integers, little changes except that the 'number' description given above is prefixed by 245 each time, this being the 'token' for the unary minus function.

Where 'n' is Floating Point

If we have 'n' as a decimal number, such as 0.234, then it's clear that the representation of 'number' is going to be in the 5 byte Floating Point format, prefixed by 31 to indicate an FP constant. If it is negative (-0.234, for example) then the 31 is prefixed by 245, as above.

Where 'n' is a numeric variable

Take the line:

```
3 y=e
```

where 'e' has already been assigned a value. The coding of the line is:

```
lenlow,lenhigh, 3,0, t1,t2,t3, 249,239, ta1,ta2,ta3, 229,0
```

Here, ta1 identifies the type of variable. As before, both variable names are stored with ASCII code of the last character of their name

increased by 128. Irrespective of the value of e, t1 will be 13 and ta1 will be 13. Typical results are:

t1	13	ta1	13
t2	14	ta2	5
t3	0	ta3	0

If we change 'e' to 'e%', then the new t values are:

t1	13	ta1	2
t2	11	ta2	5
t3	0	ta3	0

This indicates that the value 2 is used as an 'Integer' type identifier. What about array variables on the right hand side of this assignment? Let's start with the simple array e(n), which we DIM in the usual fashion, initialise to a value and then put as line 2:

```
2 y=e(6)
```

then we get t values as follows:

t1	13	ta1	13
t2	5	ta2	7
t3	0	ta3	0

In addition to the t values, we must also consider the representation of the 'e(6)'. Well, this is very simple; after the variable name comes the ASCII code for '(', then the subscript, represented in the same way as numeric constants were above, then the ASCII code for ')'.If we had an Integer array, then:

t1	13	ta1	2
t2	5	ta2	7
t3	0	ta3	0

Again, the subscript is stored in the encoded form we've already met. A two or three dimensional array seems to make no difference to 't1' value, the only change coming in the representation of the e(n,m) form in the line; naturally, the coded subscripts are separated from each other by a ','. If the subscript is a variable, then it is preceeded by three bytes to identify it's type.

Where 'n' is an expression

This appears to be fairly straight forward, and the line:

1 y=2+4

is encoded as:

lenlow,lenhigh, 1,0, 13,5,0, 249,239, 16, 244, 18, 0

where 244 is the token for '+', and the numbers 2 and 4 are held in a coded form. Should variables be involved in the expression, then they are preceeded by the ta bytes as we've already seen. If functions are called, such as SIN, then the token for the function is prefixed by 255.

If the assignment is to be made to y% instead of y, then much of the above is still true, the only difference being in the 't1' byte. t1 is now 2 instead of 13. All else appears to be the same.

Longer Variable Names

It s already been mentioned that the last character of a variable name has 128 added to it's ASCII code in order to indicate that it is the last character. Well, that's really all there is to say about these variables; there is no difference in the type bytes.

Assignments to an Array

What about the line:

y(6)=5

Well, this is tokenised with a t1 of 13 indicating FP and a ta1 byte depending upon the right hand side of the assignment. If we have typical examples, and they vary in value for different assignments. This seems to indicate that the two bytes that follow the type identifier byte are pointers to the location in memory where the data regarding the variable is stored. This possibility is further reinforced by the fact that t2 and t3 for variables that have NOT been assigned and yet are being used after the '=' sign are set to 0. This is to be expected; the BASIC interpreter has no address for such a variable. However, this is no use to us, as if we want this sort of information we simply use the '@' function!

To summarise the earlier notes

Variables are preceeded by three bytes, the first the type identifier and the others possibly a pointer in to memory. 13 indicates a FP variable, and 2 indicates an integer variable.

Numeric constants are stored in a variety of ways depending upon whether they're big integers, small integers or FP numbers.

Before we go on to look at string variables, let's take a look at a simple example of a program that 'writes' part of itself. Type in the below listing, taking care with the DATA statement.

```
 10 REM
 20 REM *** line 10 has 20 spaces ***
 30 PRINT Y
 35 GOTO 10
 40 RESTORE
 45 FOR I=372 TO 382
 50 READ A:POKE I,A
 70 NEXT I
 80 GOTO 10
 90 REM *** DATA for 'y=y+6' ***
100 DATA 13, 0, 0, 217, 239, 13, 0, 0, 217, 244, 20
```

GOTO 40 to execute the program, then list it again. Lo and Behold, line 10 has been miraculously transformed! Y will now be printed and incremented. Let's see how this technique has worked.Firstly, line 10 has a REM statement that is long enough to hold all the bytes that our 'new line' will contain; it doesn't matter if it's too long, as the spaces will be ignored by the Interpreter. The '372' in line 40 is the address of the REM token in line 10 provided that this is the first line in the program. Later we'll see how we can put REM statements that will later be converted into statement lines anywhere in the program. The FOR...NEXT loop then POKEs into the REM statement the bytes needed to make up the statement 'Y=Y+6' from the DATA statement. The bytes needed were arrived at in the following fashion.

1. 'Y' is a FP variable, so the first three bytes will be 13,0,0. The interpreter will take care of t2 and t3.

2. Next the ASCII code of 'Y', plus 128. Then the '=' token.

4. So, the next three bytes will be to indicate a Floating Point variable: 13,0,0. Then the variable name, Y (plus 128).

5. Next comes the '+' token, 244.

6. Finally, the encoded form of '6', which, from the above, is 6+14=20.

You might like to try other assignments in the DATA statement. In this routine, the values are POKEd in to the REM statement before the variable concerned is used at all in the program, but a line such as:

75 Y=90

can be added, and the routine will work perfectly starting off from 90. The only thing to take care of is the number of bytes in the DATA statement. Here are a couple more to start you off:

Y=Y*3 13,0,0,217,239,13,0,0,217, 246,17
Add line 75 Y=3, and GOTO 40 to start it off.

Y=A+Y 13,0,0,217,239,13,0,0,193,244,13,0,0,217
Add line 75 A=1, and GOTO 40 to start it off. Also, adjust the FOR...NEXT loop in line 45 to suit the number of bytes in this line.

It appears that the only 't' parameter that we need set up when we are 'entering' program lines in this way is the first one. By the way, replacing the first 't' parameter with 2 will result in the automatic insertion in the listed 'new' line of '%' signs at the appropriate places.

We'll see more examples of self modifying BASIC programs later. Let's now move on to look at string variable storage.

String Variable Assignments

The most obvious example is:

1 y$="hello"

The structure of such a line could be:

lenlow,lenhigh, 1,0, 3,5,0 249, 239, 34, text bytes, 34,0 where 249 is the modified single character variable name, 34 is the ASCII code for '"', and the t1 byte of 3, apparently indicating a string type.

For situations like:

1 y$=x$

where x$ has been previously assigned a value, the line is as follows, using the format we outlined for numeric variables:

t1	3	ta1	3
t2	12	ta2	5
t3	0	ta3	0

For the line:

```
1 y$=x$(5)
```

assuming that the array has been dimensioned, we have:

t1	3	ta1	3
t2	5	ta2	7
t3	0	ta3	0

Similarly,

```
1 y$(1)=x$
```

assuming a dimensioned array code as:

t1	3	ta1	3
t2	7	ta2	5
t3	0	ta3	0

The array subscripts are stored as for numeric arrays.Again, in each case, t2,t3,ta2 and ta3 act as pointers in to memory.

We won't look at all the constructs available in Amstrad BASIC. That would take too long and would be intensely boring, unless you appreciate that sort of thing. We will, however, consider a few others now and when we put the 'self modifying program' idea in to action.

PRINT

There are many variations on the simple PRINT command, and here we'll look at just a couple of them.

PRINT variable, constant or expression

The statement consists of the PRINT token, the space following it and then the type identifier of the variable to be printed. The t2 and t3 bytes point to the variable in memory, and then comes the variable name coded as above. If we're dealing with a constant or expression, then the rules outlined above apply.

IF...THEN

Again, this is fairly straightforward. The tokens of interest are 161 for IF and 235 for THEN. Expressions after the IF are tokenised in the way we've already seen. Thus a line like:

```
1 IF A=0 THEN PRINT
```

would be tokenised as:

lenlow,lenhigh, 1,0, 161,32, 13, t2,t3, 193, 239, 14,32, 235, 32, 191,0

We're now in a position to look at another example of a program that writes part of itself.

A simulated EVAL function

A nice function is offered on some micros by which the user can type in something like:

```
a$="2+3*SIN(4)":PRINT EVAL(a$)
```

and see the result of the expression printed to the screen. Thus expressions can be typed in in response to prompts from the program, stored as variables, and evaluated when necessary. This sort of programming makes writing data base routines rather easy. For example, say we have a simple data base whose records consist of separate strings- say n$(n), a$(n) and t$(n), which hold the Name, Address and Telephone Numbers of N people respectively. Say we wanted to find all people whose names were Smith and who lived in Croydon. If we had an EVAL function, we could assemble a string like:

```
"n$(I%)=""Smith"" AND a$(I%)=""Croydon"""
```

or something similar. Then we could use a routine such as:

```
100    FOR I%=1 TO 100
110    IF EVAL(a$)=TRUE THEN GOSUB 1000
120    NEXT I%
```

to search 100 records. GOSUB 1000 would simply print out the records containing both Smith and Croydon. Obviously, the search criteria in a$ could be as complex as you like, and because a$ can be

set up like any other string variable, allows an infinite number of different searches to be carried out with the same subroutine. All that you need to do is change a$.

However, Locomotive BASIC hasn't got such a function; but, we'll soon change that. We'll write a routine that accepts a string like that above, and uses it to put together an IF...THEN statement dynamically, while the program is running, and uses this statement to test to see if our expression is true, setting a flag to indicate that it is or isn't.

Our first task is to put a REM statement in the program that contains enough spaces to hold any expression that you'll want to evaluate; if the REM statement isn't long enough, then a long expression will probably crash the BASIC program by overwriting the Terminator byte and the next line! To make things easy, I usually put this line as close to the start of the program as possible:

```
 1 GOTO 10
 2 flag%=0
 3 REM       *** as long as needed ***
 4 RETURN
 5
10 REM **** start of program ***
```

The line 1 GOTO jumps around our 'EVAL' subroutine. Line 2 initialises a flag to 0, and the line 3 REM statement will eventually contain our expression for evaluation. The line 4 RETURN allows us to call this routine as a subroutine after we've set up line 3.

The first fact we need is the address of the REM token on line 3. If the above is typed in exactly as shown above, then this address is 396. So, the first token of our expression will be POKEd in at address 396. We will also need the length of the REM statement, from the REM token to the last space-DO NOT include the Terminator byte. We will use this last fact to ensure that we don't over run the space available, and also to allow us to set the line to spaces before new expressions are POKEd into the line.

Let's begin by seeing how we can build an IF...THEN statement and use it to set 'flag%'. The program below (Listing 1) shows how an IF statement can be poked in to memory.

```
1 GOTO 10
2 flag%=0
3 REM
4 RETURN
```

```
    5 :
    6 :
   10 INPUT"Give a value for a$";a$
   20 GOSUB 10000
   30 PRINT flag%
   40 GOTO 10
   50 :
   60 :
10000 REM subroutine to set up IF...THEN statement
10010 RESTORE
10020 GOSUB 10600
10030 FOR I%=396 TO 419
10040 READ byte%
10050 POKE I%,byte%
10060 NEXT I%
10070 GOSUB 2
10080 RETURN
10090 :
10100 :
10600 REM subroutine to put REM statement back
10610 FOR I%=397 TO 427
10620 POKE I%,32
10630 NEXT I
10640 POKE 396,197
10650 RETURN
10660 :
10670 :
65000 DATA 161, 32, 3, 0, 0, 225, 239, 34, 65, 65, 65, 34, 32,
      235, 32, 2, 0, 0, 102, 108, 97, 231, 239, 15
```

The IF...THEN statement held in line 65000 is:

```
IF a$="AAA" THEN flag%=1
```

Of course, you knew that, didn't you, by decoding the bytes in the DATA statement.

In case you weren't sure of how I got this DATA statement, here is a quick explanation.

161 is the IF token, then a space, then the string type identifier, (3,0,0). Next comes ASC("a")+128, the '=' token and the string "aaa". Finally we have the THEN token, the integer type identifier (2,0,0), the variable name 'flag', with the 'g' suitably modified, then the '=' token and the representation for '1' to finish things.

The FOR...NEXT loop beginning at line 10030 POKEs these bytes in to the 40 space REM statement at line 3. The subroutine at line 10600 sets line 3 to be a REM statement again, and replaces the first thirty characters after the REM token with spaces. If you alter this program, and put bytes in the DATA statement to represent other IF...THEN statements, you'll have to alter this routine if the statements are more than 30 characters long. In addition, of course, you'll have to alter the line 10030 FOR...NEXT loop.

RUN the program, and type in a string in response to the question. For all strings except 'AAA' flag% will be printed as 0, which is what we expect. LIST the program and you will see line 3 suitably modified. Listing 2 is a much more elaborate subroutine, which will accept a string containing an expression to be evaluated, convert it in to to the correct bytes, and POKE these bytes into line 3. The string should contain either an IF...THEN statement or a variable assignment. Examples will be given after the listing. The subroutine, as it stands, is a little limited, but you will no doubt be able to expand it.

```
    1 GOTO 50020
    2 flag%=0
    3 a%=
    4 RETURN
50000 :
50010 :
50020 INPUT s$
50030 GOSUB 50830
50040 GOSUB 50060
50050 END
50060 REM simple interpreter for
50070 REM new program lines
50080 REM variable names single
50090 REM character and identifier
50100 REM integer constants only
50101 REM will handle (var)=(expression)
50102 REM or IF (expr.) THEN (var)=(expr.)
50110 DEF FNchar(p%)=ASC(MID$(s$,p%,1))
50115 address%=396
50116 pointer%=1
50120 IF LEFT$(s$,2)="IF" THEN GOTO 51007
50130 address%=396
50140 pointer%=1
50150 GOSUB 50200
50160 GOSUB 50340
50170 GOSUB 50410
50180 RETURN
```

```
50190 :
50200 REM variable handler
50210 varnam%=FNchar(pointer%)
50220 varnam%=varnam%+128
50230 pointer%=pointer%+1
50240 type%=0
50250 IF FNchar(pointer%)=ASC("%") THEN type%=2
50260 IF FNchar(pointer%)=ASC("$") THEN type%=3
50270 IF type%=0 THEN type%=13:pointer%=pointer%-1
50280 POKE address%,type%
50290 POKE address%+1,0
50300 POKE (address%+2),0
50310 address%=address%+3:pointer%=pointer%+1:POKE
      address%,varnam%:address%=address%+1
50320 RETURN
50330 :
50340 REM = subroutine
50350 IF FNchar(pointer%)<>ASC("=") THEN GOTO 50993
50360 POKE address%,239
50370 address%=address%+1
50380 pointer%=pointer%+1
50390 RETURN
50400 :
50410 REM expression subroutine
50420 IF pointer%>LEN(s$) THEN GOTO 50510
50421 IF FNchar(pointer%)=32 THEN GOTO 50510
50425 IF FNchar(pointer%)=ASC("=") THEN GOSUB 50340:GO
      TO 50420
50430 IF FNchar(pointer%)>=ASC("A") THEN GOSUB 50200:GO
      TO 50420
50440 IF FNchar(pointer%)=ASC("+") THEN GOSUB 50530:GO
      TO 50420
50450 IF FNchar(pointer%)=ASC("-") THEN GOSUB 50580:GO
      TO 50420
50460 IF FNchar(pointer%)=ASC("*") THEN GOSUB 50630:GO
      TO 50420
50470 IF FNchar(pointer%)=ASC("/") THEN GOSUB 50680:GO
      TO 50420
50475 IF FNchar(pointer%)=ASC("(") OR FNchar(pointer%)=
      ASC(")") THEN GOSUB 51000:GOTO 50420
50480 IF FNchar(pointer%)>=ASC("0") AND FNchar(poin
      ter%)<=ASC("9") THEN GOSUB 50730:GOTO 50420
50490 IF FNchar(pointer%)=34 THEN GOSUB 50900:GOTO 50420
50491 IF FNchar(pointer%)=ASC(">") OR FNchar(poin
      ter%)=ASC("<") OR FNchar(pointer%)=ASC("|") THEN
      GOSUB 50975
50500 GOTO 50420
```

```
50510 RETURN
50520 :
50530 REM + subroutine
50540 POKE address%,&F4
50550 pointer%=pointer%+1:address%=address%+1
50560 RETURN
50570 :
50580 REM - routine
50590 POKE address%,&F5
50600 pointer%=pointer%+1:address%=address%+1
50610 RETURN
50620 :
50630 REM * routine
50640 POKE address%,&F6
50650 pointer%=pointer%+1:address%=address%+1
50660 RETURN
50670 :
50680 REM / routine
50690 POKE address%,&F7
50700 pointer%=pointer%+1:address%=address%+1
50710 RETURN
50720 :
50730 REM number routine
50740 temp%=pointer%
50750 temp%=temp%+1
50760 IF temp%<=LEN(s$) THEN IF FNchar(temp%)>=ASC("0")
      AND FNchar(pointer%)<=ASC("9") THEN GOTO 50750
50770 temp%=temp%-1
50780 sub$=MID$(s$,pointer%,(temp%-pointer%+1))
50790 num%=VAL(sub$):pointer%=temp%+1
50800 IF num%<10 THEN POKE address%,(num%+14):ad
      dress%=address%+1:RETURN
50810 IF num%<=32767 THEN POKE address%,26:POKE
      address%+1,num% MOD 256:POKE address%+
      2,INT(num/256):address%=address%+3:RETURN
50820 :
50830 REM spaces routine
50840 FOR i=396 TO 426
50850 POKE i,32
50860 NEXT i
50870 POKE 396,197
50880 RETURN
50890 :
50900 REM string reading
50910 POKE address%,34
50920 address%=address%+1:pointer%=pointer%+1
50930 POKE address%,ASC(MID$(s$,pointer%,1))
```

```
50940 IF PEEK(address%)<>34 THEN GOTO 50920
50950 address%=address%+1:pointer%=pointer%+1
50960 RETURN
50970 :
50975 REM relational operators
50976 IF FNchar(pointer%)=ASC("|") THEN GOTO 50988
50981 IF FNchar(pointer%)=ASC(">") THEN POKE address%,
      238:pointer%=pointer%+1:address%=address%+1:RETURN
50982 IF MID$(s$,pointer%,2)="<>" THEN POKE address%,
      242:address%=address%+1:pointer%=pointer%+2:RETURN
50983 IF MID$(s$,pointer%,2)=">=" THEN POKE address%,
      240:address%=address%+1:pointer%=pointer%+2:RETURN
50984 IF MID$(s$,pointer%,2)="<=" THEN POKE address%,
      243:address%=address%+1:pointer%=pointer%+2:RETURN
50985 IF FNchar(pointer%)=ASC("<") THEN POKE address%,
      241:pointer%=pointer%+1:address%=address%+1:RETURN
50986 GOTO 50993
50987 :
50988 pointer%=pointer%+1
50989 IF MID$(s$,pointer%,2)="OR" THEN POKE address%,
      &FC:address%=address%+1:pointer%=pointer%+2:RETURN
50990 IF MID$(s$,pointer%,3)="AND" THEN POKE address%,
      &FA:address%=address%+1:pointer%=pointer%+3:RETURN
50992 :
50993 REM error handler
50994 MODE 1
50995 PRINT "Ooops, error in:"
50996 PEN 3:PRINT s$
50997 PEN 1:END
50999 :
51000 REM brackets routine
51001 POKE address%,FNchar(pointer%)
51002 address%=address%+1:pointer%=pointer%+1
51003 RETURN
51004 :
51007 REM routine to build IF...THEN
51008 REM statements.
51009 POKE address%,161:address%=address%+1:pointer%
      =pointer%+3
51010 GOSUB 50410:pointer%=pointer%+1:IF
      MID$(s$,pointer%,4)="THEN" THEN POKE address%,&EB:ad
      dress%=address%+1:pointer%=pointer%+5
51011 GOSUB 50200
51012 GOSUB 50340
51013 GOSUB 50410
51014 RETURN
```

The routine is reasonably straightforward, and its limitations are given in the REEM statements. However, the routine will put together in to line 3 some reasonably complicated expressions and assignments. Note that AND and OR are supported, but must be preceded by a '|' symbol. Run the program, and type in a few simple assignments or IF...THEN statements to be inserted in line 3. The routine isn't terribly nice when it comes to errors; if you type in a silly statement, then the routine probably won't tell you. Perhaps you could add some error checking? In addition, you could also add array variables and long variable names. However, the routine should give you the right ideas.

To test the routine, try some simple expressions:

```
a%=123
```

Enter this, then, after 'Ready' returns execute:

```
a%=0
GOSUB 2
PRINT a%
```

The correct value, 123, will be printed. Similarly, IF...THEN statements such as:

```
IF a%=0 THEN g%=1
```

can be tried, or:

```
IF (a%+2)=7 THEN f$="Hello"
```

Try a few of these for yourself.

For more complex expressions in the IF...THEN statement, as you might find if you were searching a database, things like:

```
IF (a$="Smith" |AND c$="Croydon") THEN f%=1
```

could be put in to s$ and passed to the subroutine. To execute a search of a string array, therefore, we'd simply put array elements in to simple string variables before calling our 'tailor made' IF...THEN statement.

Other possibilities for 'building' program lines in this way might be routines that put DATA statements together, drawing programs which put together lines of DRAW,PLOT statements, etc. or programs which put SYMBOL definitions in to program lines. This whole book could consist of ways of putting such lines together; as I intend to go on to

different subjects, I suggest that you use a simple routine like Listing 3 to inspect program lines so that you can study their construction.

In addition, if you put together several program lines you'll need a means of finding the address of the REM token in a given program line; such a routine will be given in Chapter 4.

```
1000  FOR I%=368 TO 40000
1010  PRINT I%,PEEK(I%);"   ";
1020  IF PEEK(I%)>31 AND PEEK(I%)<127 THEN PRINT
      CHR$(PEEK(I%)) ELSE PRINT
1030  NEXT I%
```

An alternative approach to putting program lines together is much 'cleaner', in that we don't have to POKE bytes in to the BASIC program, but has the disadvantage of putting the program lines in to a disc or cassette file rather than directly in to a program. The new lines are stored in ASCII format, and so can be loaded in to the machine when needed. This approach is good for SYMBOL definitions, etc. So, we'll take a brief look at this approach to programs that can write other programs.

The application is very trivial, but will again serve to demonstrate the principles involved. The program is listed in Listing 4, and simply allows you to type in lines of BASIC in response to the '?' prompt. Up to 9 lines can be typed in to this routine, and these are stored in the array 'bline$()'. Obviously, the program could be extended by making this array larger. When you've finished, type in '`' on its own as response to the prompt. The program will then ask you for a file name and save the 'program' to tape. It can now be 'MERGEd' in to the machine, with the line numbers that you used when typing the lines in in response to the prompts. This approach is very useful for providing MERGable character and Envelope definitions, etc.

```
 10 REM program to write programs
 20 REM to a disc or tape file
 30 :
 40 DIM bline$(10)
 50 bline%=1
 60 :
 70 INPUT bline$(bline%)
 80 IF bline$(bline%)="" THEN bline%=bline%-1:GOTO 110
 90 bline%=bline%+1
100 IF bline%=10 THEN GOTO 110 ELSE GOTO 70
110 PEN 3:INPUT"What is the file name: ",a$
```

```
120 OPENOUT(a$)
130 FOR i%=1 TO bline%
140 PRINT#9,bline$(i%)
150 NEXT
160 CLOSEOUT
170 END
```

Variable Structure

We'll round off this Chapter on the storage of BASIC programs with an examination of the way in which the different variable types are stored in the computer. BASIC variables are stored after the program, and the address of variables can be easily obtained by use of the '@' command. For example:

```
fred%=2
PRINT @fred%
```

This will return the address of the value stored in the variable 'fred%'. For other numeric variables, @ can be used to find the value of the variable in memory; for string variables, the @ function returns to us the address of a three byte entry in memory which points us to the string in memory. More details about this will be given later.

Integer Variables

This is a logical place to start, because these variables are fairly simply stored. Take for example 'a%'.

```
        0
        ASC("A")+128
        1
@a%     low byte of value   value=low+
        high byte of value  256*high
        0
```

We see that in this case, the variable name is a single character, and this is stored with 128 added to it s ASCII code, in the same way that such variable names are stored in the program text. One thing to note is the way in which the letters of the variable name are stored as the ASCII codes of the Upper Case versions of the letters. This process also affects the ASCII codes of numbers in variable names. To get the name back to 'normal' use AND each character with 127 and OR if with 32. For a longer variable name, only the last character has 128 added to it s ASCII code. The storage of, say a% as A%, explains why Locomotive BASIC cannot distinguish between such variable names.

Note where @a% points to in this variable. POKEing this address, and the next one, allows you to alter the value of the variable directly. (A rather pointless exercise!)

Integer Arrays are stored in the following fashion; let's start with a%(n).

```
0
ASC("A")+128
1

numlow
numhigh

numdim

numellow
numelhigh

ele0low
ele0high

ele1low
ele1high

 ...

  0
```

Right, let's explain some of the terms used above. 'numlow' and 'numhigh' constitute a 16 bit number that is the number of bytes used to hold the array elements, number of dimensions and number of elements. For a%(10) this is 25. 3 bytes are used for the number of elements, 1 for the number of dimensions and 22 for the array elements. (a%(10) would have 11 elements, numbered 0 to 10, each element using 2 bytes.) The elements are then stored in the usual 'low-high' order.

For a multi dimensional array, such as a%(x,y), we have a similar arrangement. This is shown below.

```
0
ASC("A")+128
1
```

```
numlow
numhigh

numdim

numeleylow
numeleyhigh

numelexlow
numelexhigh

ele0,0low
ele0,0high

ele1,0low
ele1,0high

ele2,0low
ele2,0high
  ...
  0
```

'numeleylow' and 'numeleyhigh' and 'numelexlow' and 'numelexhigh' hold the number of the 'x' and 'y' elements in the array. Again, remember that arrays are numbered from 0 upwards.

'ele0,0low'... and so on are the bytes used to store the array elements. You might like to look at arrays containing various values using a simple program to print out the value of the bytes that make up the array structure; after all, that's how I worked out the details given here!!

Strings

String variables are the next easiest variable type to consider; we'll leave Floating Point Variables until last, because they are a little complicated.

The '@' function, when applied to a string variable returns the address of a three byte area in memory called the String Descriptor. Take the below example, for the program statement a$="Hello".

```
0
ASC("A")+128
2                    String Type
                     Identifier
```

5	Descriptor
	Block byte 1
addlow	Byte 2
addhigh	Byte 3

byte

Again, the variable name is stored with it's last character modified by the addition of 128 to it s ASCII code. '2' is used as the string type identifier here; dare I say it would have been a little easier for us if the Type Identifiers used here and in variable definitions in the program text had been the same? The String Descriptor starts with the string length, in this case 5, and then continues with the address at which the first character in the string can be found in memory. The address will be either in the body of the program, if the definition of the string is like any of the following,

```
1000  fred$="Hello"

 110  f$="test string"
 120  a$=f$
```

or will be high in memory, if the definition of the string was made as a direct command, such as

```
a$="fred" <ENTER>
```

or was made by concatenating strings or using CHR$. Thus the characters in a string can be directly accessed by using @.

String Arrays

These are stored in memory as a collection of String Descriptor blocks, with a 'preamble' that is similar in structure to that for Integer Array. The structure for the array a$(n) is shown below.

0
ASC("A")+128
2

lenlow
lenhigh

dims

elelow
elehigh

descriptor for
element 0

'lenlow' and 'lenhigh' hold the number of bytes used to hold the String Descriptors, number of elements and number of dimensions.

'dims' holds the number of dimensions in the array, and 'elelow' and 'elehigh' hold the total number of elements in each dimension of the array. Again, remember that array elements are numbered from 0 upwards. This two byte entry is repeated for each byte of the array. So, for example, for the array a$(n,m), the first two byte entry after 'dims' is the number of elements in the 'm' dimension, followed by a two byte entry holding the 'n' entry.

Two dimensional arrays have their elements in the following order:

0,0 1,0 2,0 3,0 ....

Floating Point Numbers

And so to the fun packed world of Floating Point (FP) numbers. As we've already briefly mentioned, they are stored in 5 bytes and are the means by which the Amstrad represents fractional or very large numbers.

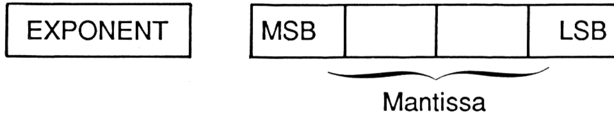
You can probably see that, using a straight binary representation of numbers, 5 bytes wouldn't hold very large numbers. What we in fact do is hold the number as a MANTISSA and EXPONENT. In decimal arithmetic, we can write 1000 as:

1 * 10 E+3

Here, 1 is the mantissa and 10 EXP(3) is the Exponent. Similarly, we could write 1234 as:

1.234 * 10 E+3

The mantissa represents the value of the number, and the Exponent indicates the position of the decimal point within that number. Well, in binary it's the same, except we've got a binary point to worry about rather than a decimal point. The arrangement of the 5 bytes is:



The Mantissa holds the binary representation of the number, without any binary point needed. The position of the binary point is reflected in the value of the Exponent.

Now, if we consider the binary number:

11.101101

then all the following combinations of Mantissa and Exponent represent it.

```
11.101101    * 2 E 0
 1.1101101   * 2 E +1
 0.11101101 * 2 E +2
```

In computers, we generally prefer a single representation, and so over the years a 'standard' means of representing Floating Point numbers has been developed. This is known as Standard Form, or Normal Representation. The process of getting an FP number into this format is called 'Normalisation'. The first thing to note is that this format assumes that the binary point is always to the left of the Most Significant Bit of the Mantissa. The mantissa will, therefore, always represent a binary fraction in this format, the Exponent indicating the position of the binary point within the actual number being represented.

The process of producing a normalised FP number is as follows. To make things concrete, let's turn 1.0 into a Normalised FP Number.

i/ Write down a representation of the number in binary. This will be the Mantissa. So, we could have:

0000 0001 0000 0000 0000 0000 0000 0000

It doesn't matter which byte of the Mantissa we put the 1 in to; after all, the value in the Exponent can be used to modify the position of the Binary point.

ii/ Get a value for the exponent which positions the Binary Point in the Mantissa. In this case, remembering that the binary point starts off to the left of the MSB of the mantissa, the Exponent will be 8. This gives us:

0000 1000 0000 0001 0000 0000 0000 0000 0000 0000

iii/ Now we Normalise the number. We do this by shifting each bit of the mantissa to the LEFT until the MSB of the mantissa has the value '1'. Simple binary arithmetic tells us that for each bit position we move to the left, we increase the value of the number by a power of two. It follows therefore, that to keep the number representing the same value we must decrease the value of the exponent by 1 for each bit position that we move to the left. For the number above, we end up with:

0000 0001   1000 0000 0000 0000 0000 0000 0000 0000

That completes normalisation.

iv/ We now come to consider the sign of the number. There are two approaches that can be used here. We could use a separate bit or byte to hold the sign of the number, or we could somehow incorporate it in to the 5 bytes we're already using. Well, the latter approach is chosen. Any normalised FP number is expected to have the MSB of the mantissa set to 1, and so we can use this bit as the sign bit. If the number is positive we set it to 0 and if negative we set it to 1. This has the added advantage of giving us a quick way of changing the sign of a number; we simply add 128 (&80) to the most significant byte of the mantissa. Applying this to our representation so far we get:

&01 &00 &00 &00 &00

v/ Finally, we add &80 to the exponent byte. This gives us:

&81 &00 &00 &00 &00

and that's it. What about fractional numbers, such as 0.25? The same rules apply. Firstly, write down a mantissa and exponent that represent the number. For example:

0000 0000   0100 000 0000 0000 0000 0000 0000 0000

Now, the binary point is to the left of the MSB of the mantissa in any case, so in this example the exponent starts off as zero.

To normalise the representation, the mantissa must be shifted bitwise to the left until the MSB is 1. So, we end up with:

-1   1000 0000   0000 0000 0000 0000 0000 0000

Now we set the MSB of the mantissa to zero, as we had a positive number, and add &80 to the Exponent to finish things off. This gives us:

```
&7F  &00  &00  &00  &00
```

There is one special case in this method of representing numbers; this is 0, which is stored with the Exponent and all the mantissa bytes set to 0.

As to the way in which FP variables are stored in the Amstrad, the below bytes represent the variable a=0.25.

```
        0
        ASC("A")+128
        4           Type

@ points
here    0           LSB Mantissa
        0
        0
        0           MSB Mantissa
      &7F           Exponent
        0
```

```
 10 REM demonstration of Real Number
 20 REM structure on Amstrad
 30 :
 40 MODE 1
 50 :
 60 INPUT "What is the number",a
 70 i%=0:a%=0:a%=@a
 80 PRINT:PRINT
 90 PEN 3:PRINT "Mantissa LSB   ";:PEN 1:PRINT
    HEX$(PEEK(a%+0))
100 PRINT SPC(15);HEX$(PEEK(a%+1))
110 PRINT SPC(15);HEX$(PEEK(a%+2))
120 PRINT SPC(15);HEX$(PEEK(a%+3))
130 PEN 3:PRINT "Exponent LSB   ";:PEN 1:PRINT
    HEX$(PEEK(a%+4))
140 PRINT:PRINT
150 GOTO 60
```

The bytes making up the FP representation are stored 'backwards' in memory. Floating Point arrays are stored in a similar fashion to Integer Arrays, though with 5 byte FP entries replacing the two byte Integer entries.

If you are interested in the way in which FP numbers are stored in memory, then the program in Listing 5 might be of use. It prints out the bytes used to represent any value you type in. To use, simply RUN it and enter the value when prompted. @ is used to get the address of the five bytes used.

That completes this brief review of Amstrad variable structure. In Chapter 4 we'll come across routines that use the Information presented in this Chapter to produce some useful utilities for BASIC programmers that can be accessed by the RSX system. However, I'd like to finish this Chapter off with a few comments about speed and cutting down the size of programs.

Program Speed

The usual answer to any one who asks "How can I make my BASIC program go faster?" is "Write it in machine code!".

However, there are techniques that can be applied to any BASIC program to give it a little more speed.

Use of Integer Variables

If integer variables are used wherever possible, a program will run faster. For example, the line:

```
FOR i%=1 TO 1000:NEXT i%
```

will run about twice as fast as:

```
FOR i=1 TO 1000:NEXT i
```

The reason for this is clear; the interpreter has less work to do in sorting out Integers than in decoding Floating Point numbers. However, there will be some situations in which Floating Point numbers are necessary. However, in some of these cases scaling might be useful. For example, let's assume we're writing a program dealing with money. We might have sums of money like £1.22 to deal with. Well, we could simply convert such sums in to pence by multiplying by 100, giving 122. We could then operate on these integers until we come to display the result. However, this would only be useful for small sums of money, due to the limited range of numbers that can be held in an integer variable.

FOR...NEXT loops

We've already seen that an integer variable makes the FOR...NEXT loop go faster. There are a couple of other things to try with these loops to speed them up a little. The first is to miss off the variable name off of the NEXT wherever possible. This results in a slight speed up of the loop. The second thing to consider is to try and put the statements to be executed by the FOR...NEXT loop on the same line of the program e.g:

```
100     FOR i=1 TO 100:PRINT i:NEXT
```

rather than:

```
100     FOR i=1 TO 100
110     PRINT i
120     NEXT
```

Functions

A BASIC function or arithmetic operation takes a finite amount of time, and some functions are very long winded indeed. The below table gives a rough ranking of the time taken by some functions and arithmetic operations.

FAST	+,-
	*,/
	ASC()
	SQR()
	SIN,COS
	^
SLOW	TAN

The only thing to do is to cut down the number of times that the more complex functions, such as TAN, are evaluated.

Wherever possible, such operations should be done outside loops. TAN takes about twice as long as SIN and COS. Also, the actual time taken by some functions, '*' for example, depends upon whether the numbers involved are Integer or Floating Point.

Space Considerations

When we consider the amount of RAM available to the user on the Amstrad, and compare it to the amount available on other popular microcomputers you might wonder why we bother worrying about space considerations. However, here we go.

Variables

Use short variable names wherever possible, and, again, integers if possible. This is due to integers using less space in memory than FP variables.

Multi-Statement Lines

If you get as many statements as possible on to a single line, then you will save space. This is because the bytes needed for each separate line, such as the line number, length and terminator byte will not be needed.

REM Statements

Simply remove them.

In the next Chapter, we'll look at a collection of RSX routines that will hopefully make BASIC programming easier.

Chapter 4
RSX Routines

In this Chapter you'll find a variety of extensions to the Amstrad BASIC to help you with your BASIC programming. These commands are added via the Resident System Extension facilities that are offered by the Operating System of the Amstrad. The routines are all added to BASIC by machine code programs, which will be presented in two ways in this Chapter. At the end of the Chapter you will find a BASIC Loader program which is used to POKE the bytes that make up the machine code in to memory, thus allowing you to use the commands with no knowledge whatsoever of machine code. Listings of the bytes needed will also be given. However, for those of you who are interested, I also offer Assembler Listings of the various routines so you can 'take them apart' and possibly adapt them to your own use. In addition, there will also be hints on combining machine code and BASIC.

The RSX System

Those of you who are conversant with the RSX facilities can skip this section, but if you're not, then this is what it's all about!

Quite simply, it's a means of adding 'commands' to the Amstrad. These commands can best be viewed as named CALL statements; indeed, parameters are passed to RSX statements in exactly the same way as that in which we pass parameters to machine code routines using the CALL statement.

RSX commands are prefixed by the '|' symbol, which can be accessed on the keyboard by SHIFT-@. This system is perfect for applications in which we wish to pass values from BASIC into machine code, but, like the CALL statement, is a little awkward when it

comes to passing values back, as we'll soon see. The biggest advantage of adding commands via the RSX system, though, rather than just CALLing the appropriate machine code routines is that you don't have to remember the addresses of the routines; quite handy if there are a lot of routines!

At the heart of this system are two tables. I'll give you sufficient knowledge here to allow you to add your own commands if you want to, but for full details I direct you to the 'Amstrad Technical Firmware Manual'.

The Jump Table

This table tells the OS where in memory the routines are to be found. It s structure is shown below, in standard Z-80 Assembler Language.

```
jumptable   DEFW   nametable
            JP     routine1
            JP     routine2
                   ...
```

'nametable' is the address in memory of the other table used by RSX, the Name Table, which we'll look at shortly. The rest of the table consists of a series of JP instructions. In each case, the JP is to the address of one of the RSX routines. These JPs must be in the same order as that in which the names of the routines are listed in the Name Table. Obviously, one Jump Table entry is needed for each RSX command to be added.

The Name Table

This is a table in memory that holds the names of the RSX commands to be added. The start address of this Table is stored, low byte first, in the first two bytes of the Jump Table. The structure of a typical Name Table is shown below.

```
nametable    DEFS  name-of-routine1
             DEFS  name-of-routine2
             DEFS  name-of-routine3
             ...
             DEFB  0
```

Here '0' is used to terminate the Table. The command names are stored in Upper Case with the last character of the name modified by having 128 added to it s ASCII code. The order of these routine

names corresponds to the order of JPs in the Jump Table. Thus, when the BASIC interpreter encounters '|name-of-routine1' in a program it will jump to address 'routine1' where it will expect to find a suitable machine code program to run.

In addition to these two tables, the OS also requires a 4 byte block of memory to act as workspace. This must be in the central 'Memory Pool', not overlaid by ROM. There now remains the problem of telling the BASIC Interpreter about these tables. That's easy; simply call &BCD1 with HL holding the address of the first byte of workspace and the address of the start of the JUmp Table in the BC register pair.

Only one thing; it's not a good idea to call this routine more than once for a given name/jump table combination. It occasionally causes the machine to hang up!

Parameter Passing

This is the same as for the CALL statement. On entry to the RSX routine, IX points to a Parameter Block and A holds the number of parameters that were passed over to the routine. The parameter types that can follow an RSX command are:

i/ A number or variable that gives a value in the range 0 to 65535. e.g.

```
|newcommand, 100
a%=400:|newcommand,a%
```

The value is available to the machine code program, as we shall soon see, but it's not possible to pass values back from the machine code program using this technique.

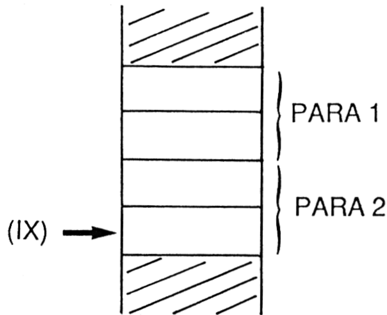
ii/ A variable preceded by the '@' symbol. This allows us to pass values from BASIC to machine code and machine code to BASIC. e.g.

```
a%=45:|newcommand,@a%
PRINT a%
```

iii/ A string variable preceded by the '@' symbol. It is not possible to pass string constants over to machine code routines in this way, only variables.

How are these variables and constants made available to the machine code programs? Well, on entry to the RSX routine IX points to a Parameter block as shown below.

```
newcommand, para1,para2
```



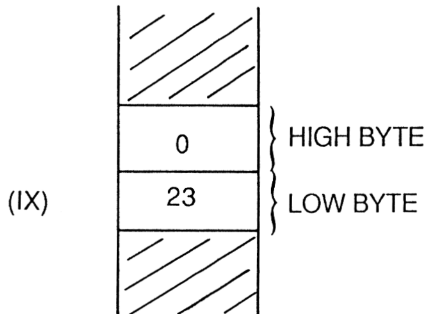
Each parameter passed sets up a two byte entry in the parameter block, the contents of these two byte entries depending upon the parameter type.

Integer Numbers or Variable

Here the two byte entry contains the binary representation of the value. Thus for the command:

```
newcommand,23
```

the parameter block would be:

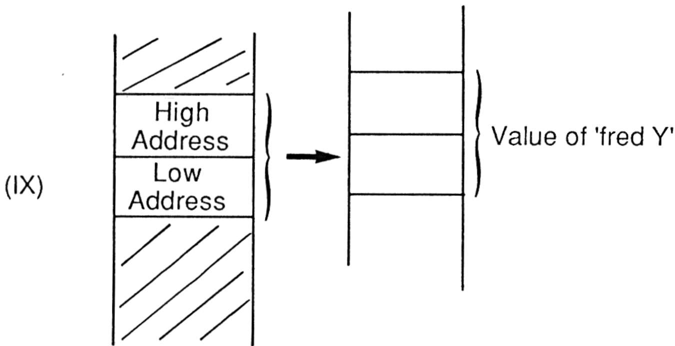


Numeric Variables prefixed by '@'.

Here, the two byte entry contains the ADDRESS of the variable. Thus for the command:

```
newcommand,@fred%
```

the parameter block:



will be set up, with the address of 'fred%' being in the parameter block. This address can be used to get the current value of the variable 'fred%'. In addition, though, the address can be used by the machine code program to alter the value of 'fred%'. Thus a machine code program can alter the value of a BASIC variable, allowing values to be passed back to BASIC from machine code. There is one important point to remember with regard to passing these '@' parameters, though, and that is that the variable used, 'fred%' in this case, must be assigned a value before it is passed over to the RSX command, even if this value is just 0. Failure to do this will generate an error.

String Variables prefixed by '@'

In this case, the two byte parameter block entry is the address of a String Descriptor Block, which was described in Chapter 3. Again, the string variable can be modified by the machine code. Also, the string variable must have a value assigned to it before it's passed to an RSX command.

If you already have an RSX extension in operation in the machine, then you can set up different RSX tables as above and use &BCD1 to activate them. This won't affect the first RSX commands, but you should ensure that two commands don't have the same name!

The MEMORY command should be used to set up a safe area in memory into which the machine code can be placed. These RSX commands were written on a 464 tape system, but there is no reason why they shouldn't work perfectly well on a disk system.

The Routines

The RSX routines will be listed in 'modules', each module being a

functionally separate part of the program. This allows discussion of the machine code as we go along. As the different sections have line numbers, the listings will be easy to enter. They were written using the Amsoft DEVPAC Assembler.

```
10 ; RSX commands, Module 1
20 ; initialises the RSX command
30 ; table and saves jumpblock
40 ; entries which the commands may
50 ; alter in use
60
70
80       ORG 38000
90 init:  LD  HL,wkspace
100      LD  BC,jpblk
110      CALL #BCD1
120
130 ; the above lines set up the RSX
140 ; commands
150
160
170      LD  A,(#BDEE)
180      LD  (brktem),A
190      LD  A,(#BDF1)
200      LD  (prt1),A
210      LD  A,(#BDF2)
220      LD  (prt2),A
230      LD  A,(#BDF3)
240      LD  A,(prt3)
250
260 ; these lines have saved the contents
270 ; of jumpblocks that will be altered
280 ; in use
290
310      RET
320 wkspac: DEFS 4
360
370 jpblk:  DEFW namtab
380      JP  reset
390      JP  pause
400      JP  find
410      JP  dump
420      JP  search
440      JP  get
450      JP  romr
460      JP  llist
470      JP  crtc
480      JP  sound
490      JP  ink
500      JP  amplit
510      JP  tone
520      JP  symbol
530      JP  brkon
540      JP  brkoff
550      JP  cat
560      JP  motor
570
580 namtab: DEFM "RESE"
590      DEFB "T"+128
600      DEFM "PAUS"
610      DEFB "E"+128
620      DEFM "FIN"
630      DEFB "D"+128
640      DEFM "DUM"
650      DEFB "P"+128
680      DEFM "SEARC"
690      DEFB "H"+128
700      DEFM "GE"
710      DEFB "T"+128
720      DEFM "ROMREA"
730      DEFB "D"+128
740      DEFM "LLIS"
750      DEFB "T"+128
760      DEFM "CRT"
770      DEFB "C"+128
780      DEFM "SOUN"
790      DEFB "D"+128
800      DEFM "IN"
810      DEFB "K"+128
820      DEFM "AMPLITUD"
830      DEFB "E"+128
840      DEFM "TON"
850      DEFB "E"+128
860      DEFM "SYMBO"
870      DEFB "L"+128
880      DEFM "BRKOF"
890      DEFB "F"+128
900      DEFM "BRKO"
910      DEFB "N"+128
920      DEFM "CA"
930      DEFB "T"+128
940      DEFM "MOTO"
950      DEFB "R"+128
960      DEFM DEFB        0
965
970 ; these are the command names
990
1000 brktem: DEFB 00
1010 prt1:   DEFB 00
1020 prt2:   DEFB 00
1030 prt3:   DEFB 00
1040
1050 ; the above bytes are for
1060 ; temporary storage of bytes
1070
1080
```

Module 1 of the program sets up the Tables for the RSX commands that we will add to the BASIC of the Amstrad.

The program has been written to be self-explanatory. Lines 170 and 180 save the current value of the Jumpblock entry that is responsible for Break event testing. Lines 190-240 save the current contents of the Jumpblock entry that usually allows the character in the A register to be sent to a printer. The|LLIST command will modify this entry, and we must have a copy of it so that we can restore the Jumpblock to normal when necessary.

This initialisation routine should only be called once, after which the commands will be active.

The program has been written to start at address 38000 (as set by the ORG command in line 80).

The rest of this module is mainly the Jump and Name Tables for the new commands. Lines 1000-1030 supply some workspace.

Module 2 is more interesting. This contains the code for some of the commands. Let's examine the code in more detail.

```
1090
1100 ; ******* MODULE 2 *******
1110
1120 ; this adds the shorter RSX commands
1133
1134 parerr: LD   HL,errmess
1135 loop1:  LD   A,(HL)
1136         CP   0
1137         RET  Z
1138         CALL #BB5A
1139         INC  HL
1140         JR   loop1
1150 errmes: DEFM "Parameter Error!!"
1160         DEFB 13
1170         DEFB 10
1180         DEFB 7
1190         DEFB 0
1200
1210 ; routine prints out the error
1220 ; message if not correct number
1230 ; of parameters.
1233
1240 brkoff: LD   A,(brktem)
1250         LD   (#BDEE),A
1260         RET
1270
1280 ; brkoff routine turns on the
1290 ; normal Escape action
1320
1330 brkon:  LD   A,201
1340         LD   (#BDEE),A
1350         RET
1360
1370 ; brkon puts a RET in #BDEE
1380
1400
1410 crtc:   CP   2
1420         JP   NZ,parerr
1430         LD   A,(IX+2)
1440         LD   BC,#BC00
1450         OUT  (C),A
1460         LD   A,(IX)
1470         LD   BC,#BD00
1480         OUT  (C),A
1490         RET
1500
1510 ; CRTC routine allows the BASIC
1520 ; programmer to access the CRTC
1610
1620 sound:  CP   2
1630         JP   NZ,parerr
1640         LD   A,(IX+2)
1650         LD   C,(IX)
1660         CALL #BD34
1670         RET
1680
1690 ; this routine uses an OS routine
1700 ; to directly access the Sound chip
1710
1730 motor:  CP   1
1740         JP   NZ,parerr
1750         LD   A,(IX)
1760         CP   0
1770         JR   Z,motoff
```

```
1780        CALL #BC6E
1790        RET
1795
1800 ; above routine turns motor on
1810
1830 motoff: CALL #BC71
1840        RET
1850
1860 ; above routine turns motor off
1870
1890 reset:  CALL #BB4E
1900        CALL #BB00
1910        CALL #BBBA
1920        CALL #BC02
1930        CALL #BC65
1940        CALL #BCA7
1950        CALL #BD37
1960        LD   A,(#BDEE)
1970        LD   (brktem),A
1971        RET
1975
1980 ; 'reset' simply sets the jumpblocks
1990 ; back to normal
2000
```

Lines 1134-1180

This subroutine is used to print out the message 'Parameter Error!!" in the event of an RSX command being entered without the appropriate number of parameters. It s operation is simple; the HL register is set up to point to the message, and then the character pointed to by HL is printed to the text VDU until a zero byte is encountered. This routine could be used as a general string printing routine if HL were set up first and the routine entered at 'loop1'.

Lines 1240-1260

These execute the |BRKOFF command by setting &BDEE to it s usual value.

Lines 1330-1350

The |BRKON command turns on the new Escape key handler. This disables the Escape key in a running program, even preventing the SHIFT-CTRL-ESCAPE sequence from operating. A Z-80 RET code is loaded in to address &BDEE.

This ensures that when a call to this routine is made by the OS an immediate return is made, rather than a jump to the appropriate OS routine to check the Escape key.

Lines 1410-1490

These deal with the |CRTC command by writing values directly to the 6845 CRTC chip. This is accessed through the I/O map of the Amstrad. The register number is written to I/O address &BC00 and the data to address &BD00. Details of registers and what they do will be given in Chapter 5. Should the number of parameters passed to the command be incorrect then a message is printed and no further action is taken.

Lines 1620-1670
This code executes the |SOUND command. The Firmware routine called via &BD34 ensures that write operations to the Programmable Sound Generator do not interfere with the normal operation of the OS. A message is printed if the number of parameters is incorrect and no further action is taken.

Lines 1730-1840
These lines deal with the |MOTOR command. If there isn't one parameter no further action is taken. Otherwise OS routines are used to turn the Cassette motor on or off as required.

Lines 1890-1971
Here a variety of initialisation routines in the ROM are called and these return much of the system back to it s initial status. Finally, the variable 'brktem' is set to the initial value of location &BDEE.

Module 3 contains a few more subroutines that will be used by other routines, and also adds 4 commands. Again, the function and syntax of the new commands will be examined after we've described the listing.

```
2010
2020 ; ******* MODULE 3 *******
2030
2090 pause:  CP   1
2100         JP   NZ,parerr
2110
2120 ; check for one parameter
2130
2140         LD   C,(IX)
2150         LD   B,(IX+1)
2160 ploop:  PUSH BC
2170         CALL #BD19
2180
2190 ; BC holds the pause length in 1/50
2200 ; of a second units
2210
2240         CALL #BB1B
2250         JR   C,finish
2260
2270 ; if key pressed then finish
2280
2290         POP  BC
2300         DEC  BC
2310         LD   A,B
2320         OR   C
2330         JR   NZ,ploop
2340         RET
2350 finish: POP  BC
2360         RET
2370
2380 ; decrement BC, and finish if zero
2390
2410 get:    CP   1
2420         JP   NZ,parerr
2430
2440 ; get parameter number
2450
2460         LD   L,(IX)
2470         LD   H,(IX+1)
2480
2490 ; get address of parameter
2500
2510         CALL #BB18
2512 ; wait for a key press
2513
2530         LD   (HL),A
2540         INC  HL
2550         LD   (HL),0
2560         RET
2570
2590 ; the following subroutines
2600 ; print messages of a given length
2610 ; and print the contents of
2620 ; HL register pair as a number
```

```
2630
2660 messag: PUSH BC
2670        LD  A,(HL)
2680        CALL #BB5A
2685        POP  BC
2686        INC  HL
2687        DEC  BC
2690        LD   A,B
2700        OR   C
2710        JR   NZ,messag
2715        RET
2720 lret:  LD   A,13
2730        CALL #BB5A
2740        LD   A,10
2750        CALL #BB5A
2760        RET
2770
2820 pnumhl: LD  DE,10000
2830        CALL p1
2840        LD   DE,1000
2850        CALL p1
2860        LD   DE,100
2870        CALL p1
2880        LD   DE,10
2890        CALL p1
2900        LD   DE,1
2910 p1:    XOR  A
2920 hloop: SCF
2930        CCF
2940        SBC  HL,DE
2950        JR   C,numout
2960        INC  A
2970        JR   hloop
2980 numout: ADD HL,DE
2990        ADD  A,#30
3000        PUSH HL
3010        CALL #BB5A
3020        POP  HL
3030        RET
3040
3050
3180 cat:   LD   A,#2C
3190        LD   DE,64
3200        LD   HL,buffer
3210        CALL #BCA1
3211        JR   C,ok
3212        RET
3220
3230 ; above code loads in a 64 byte
3240 ; block from tape
3250
3260
3270 ok:    LD   HL,buffer
3280        LD   BC,16
3290        CALL messag
3295        CALL lret
3300
3310 ; print out the file name
3320
3330        LD   HL,len
3340        LD   BC,9
3350        CALL messag
3360
3370 ; print out 'Length'
3380
3390        LD   IX,buffer
3400        LD   L,(IX+24)
3410        LD   H,(IX+25)
3420        CALL pnumhl
3421        CALL lret
3422
3423 ; print out the file length
3424
3430        LD   HL,start
3440        LD   BC,9
3450        CALL messag
3460
3470 ; print the word 'Start'
3480
3490        LD   L,(IX+21)
3500        LD   H,(IX+22)
3510        CALL pnumhl
3520        CALL lret
3521
3522 ; now print the start address
3523
3540        LD   HL,block
3560        LD   BC,9
3570        CALL messag
3580        LD   L,(IX+16)
3590        XOR  A
3600        LD   H,A
3610        CALL pnumhl
3615        CALL lret
3620        CALL typer
3630        JR   cat
3640
3650 ; above code prints out the
3660 ; block number and calls a
3670 ; subroutine to print out the
3680 ; file type. then return to cat
3690
3700 typer: LD  BC,9
3710        LD   HL,type
3720        CALL messag
3730        LD   A,(IX+18)
3740        AND  14
3750        SRL  A
3760
3770 ; get the type bits into the
3780 ; LSBs of the A register
3790
```

```
3810        CP  0
3820        JR  Z,bas
3830        CP  1
3840        JR  Z,code
3850        CP  2
3860        JR  Z,screen
3870        CP  3
3880        JR  Z,ascii
3890 cout:  CALL form
3900        RET
3910
3920 ; now print the type out
3930
3940 bas:   LD  BC,5
3950        LD  HL,basicm
3960        CALL messag
3970        JR  cout
3980 code:  LD  BC,6
3990        LD  HL,codem
4000        CALL messag
4010        JR  cout
4020 screen: LD  BC,12
4030        LD  HL,screm
4040        CALL messag
4050        JR  cout
4060 ascii: LD  BC,5
4070        LD  HL,ascm
4080        CALL messag
4090        JR  cout
4100 form:  LD  B,6
4110 floop: LD  A,13
4120        CALL #BB5A
4130        LD  A,10
4140        CALL #BB5A
4150        DJNZ floop
4170        RET
4180
4190 ; now more workspace for the
4200 ; CAT statement and string
4210 ; definitions
4220
4230 type:  DEFM "Type      "
4240 len:   DEFM "Length    "
4250 start: DEFM "Start     "
4260 block: DEFM "Block     "
4270 basicm: DEFM "BASIC"
4280 codem: DEFM "BINARY"
4290 screm: DEFM "Screen Image"
4300 ascm:  DEFM "ASCII"
4310 buffer: DEFS 70
4320
4330 ; now comes the definition of the
4340 ; |SYMBOL command
4350
4360 symbol: CP  2
4370        JP  NZ,parerr
4380
4390 ; jump if not 2 parameters
4400
4410        CALL #B906
4420        PUSH AF
4430
4440 ; page in ROM with char. definitions
4450 ; and save the ROM status on the stack
4460
4480        LD  A,(IX+2)
4490        CALL #BBA5
4500
4510 ; get char. def. into HL register
4520
4540        LD  E,(IX+0)
4550        LD  D,(IX+1)
4560        PUSH DE
4570        POP IX
4580
4590 ; get address of buffer area into IX
4600 ; register pair
4630
4640        LD  B,8
4650 sloop: LD  A,(HL)
4660        LD  (IX+0),A
4670        INC HL
4680        INC IX
4690        DJNZ sloop
4700        POP AF
4710        CALL #B90C
4720        RET
4730
4740 ; transfer the bytes into the buffer and
4750 ; restore the ROM status to what it was
4760
```

Lines 2090-2360 deal with the pause command, and use 2 OS routines. The first, at &BD19, waits for the Frame Flyback to occur. This happens, roughly, once every 50th of a second. The second routine, called at &BB1B, tests the keyboard and exits with the carry flag set if a key has been pressed. If you want further details of the Firmware routines, then consult the 'Amstrad Technical Firmware Guide'.

Lines 2410-2570 execute the |GET instruction and wait for a key to be pressed before passing the ASCII code of that key back to BASIC. Lines 2530-2570 put the ASCII code into the variable that was passed as the parameter for this command.

Lines 2660-2715 deal with the printing out of messages of a set length.

The 'lret' routine simply sends a Carriage Return to the screen, forcing a new text line.

'pnumhl' prints out the contents of the HL register pair as a 5 digit decimal number. The process is simply one repeated subtraction of powers of 10 from the number, first getting the number of ten thousands in the number, then thousands, and so on down to the number of units. The result is then converted to the ASCII code for the digit in question and printed out.

Lines 3180-4320 deal with the implementation of the |CAT command, which is an extended CAT command. The routine uses the Firmware Cassette read routine called at &BCA1 to read in the 64 byte header of each block of a cassette file. This data is put in an area of memory called 'buffer'. The actual details of this Header have no place in this book, and have been described in *The Ins and Outs of the Amstrad* published by Melbourne House.

The rest of the 'cat' routine simply accesses this buffer area and prints out the information found therein.

The final lines of code, from 4330-4760 deal with the |SYMBOL command, which returns the 8 byte pattern definition for a given ASCII code. The pattern definition is stored in an area of memory whose address is passed to the routine as one of the parameters of the command.

The fourth module simply adds 4 more commands. The first of these is |LLIST, and this alters the behaviour of the normal LIST £8 command, as will be described later. Lines 4810-5050 turn the LLIST option on by pointing the MC WAIT PRINTER jumpblock to a piece of our own machine code, 'prchar'. The lines 5070-5130 turn this option off by restoring the original contents of the Jumpblock.

'pchar' at 5150-5180 first tries to write the character to the printer; this is done by calling the original MC WAIT PRINTER routine. If this fails due to the printer being busy, then a return is made with C clear. This is detected in line 5160. If C is clear then a return to BASIC is made. Otherwise, the character is also printed to the screen.

```
4770
4780 ; ******* Module 4 *******
4790
4810 llist: CP   1
4820        JP   NZ,parerr
4850        LD   A,(IX)
4860        CP   0
4870        JR   Z,loff
4880
4890 ; decide if option on or off
4900
4920        LD   A,#C3
4930        LD   HL,prchar
4940
4950 ; now set jumpblock up
4990
5000        LD   (#BDF1),A
5010        LD   A,L
5020        LD   (#BDF2),A
5030        LD   A,H
5040        LD   (#BDF3),A
5050        RET
5060
5070 loff:  LD   A,(prt1)
5080        LD   (#BDF1),A
5090        LD   A,(prt2)
5100        LD   (#BDF2),A
5110        LD   A,(prt3)
5120        LD   (#BDF3),A
5130        RET
5140
5150 prchar: CALL prt1
5160        RET  NC
5170        CALL #BB5A
5180        RET
5190
5195 ; character to printer and screen
5200
5220 ink:   CP   2
5230        JP   NZ,parerr
5270        LD   A,(IX+2)
5280        CALL #BC35
5290        LD   L,(IX+0)
5300        LD   H,(IX+1)
5310        LD   (HL),B
5320        INC  HL
5330        LD   (HL),C
5340        RET
5350
5360 ; get colours and put in variables
5370
5450 find:  CP   2
5460        JP   NZ,parerr
5470        LD   L,(IX)
5480        LD   H,(IX+1)
5490        LD   (varadd),HL
5500
5510 ; store variable address
5530
5540        LD   L,(IX+2)
5550        LD   H,(IX+3)
5560        LD   (lineno),HL
5570
5580 ; store line number away
5590
5600        LD   IX,368
5630 findlo: LD   C,(IX+2)
5640        LD   B,(IX+3)
5650        LD   HL,(lineno)
5658        SCF
5659        CCF
5660        SBC  HL,BC
5670        JR   Z,found
5680
5690 ; test line number and if OK
5700 ; jump out of routine
5710
5720        CALL #BB1B
5730        RET  C
5740
5750 ; check for the key press
5760
5770
5790        PUSH IX
5800        POP  HL
5810        LD   E,(IX)
5820        LD   D,(IX+1)
5830        ADD  HL,DE
5840        PUSH HL
5850        POP  IX
5860        JR   findlo
5870 found: INC  IX
5880        INC  IX
5890        INC  IX
5900        INC  IX
5910        PUSH IX
5920        POP  DE
5930        LD   HL,(varadd)
5940        LD   (HL),E
5950        INC  HL
5960        LD   (HL),D
5970        RET
5980
5990 ; update IX to point to next
6000 ; line length byte then round
6010 ; again.  Found routine returns
6020 ; the address in a variable
6030
6060 varadd: DEFW 0000
6070 lineno: DEFW 0000
6080
6090 ; now the search command
6100
6110 search: CP   2
```

```
6120        JP   NZ,parerrs
6160        LD   L,(IX+2)
6170        LD   H,(IX+3)
6180        LD   A,(HL)
6190        LD   (length),A
6200
6210 ; get length into A
6220
6230        INC  HL
6240        LD   C,(HL)
6250        INC  HL
6260        LD   B,(HL)
6261        LD   (stradd),BC
6262
6263 ; string address into stradd
6264
6266        LD   A,(IX)
6267        CP   1
6268        JR   Z,sok
6269
6270 ; if mode is 1 then jump on
6271
6272        LD   A,(length)
6273        LD   E,A
6274        XOR  A
6275        LD   D,A
6276        DEC  DE
6277        LD   HL,(stradd)
6278        ADD  HL,DE
6279        LD   A,(HL)
6280        ADD  A,128
6281        LD   (HL),A
6282
6283 ; add 128 to last character if it's a
6284 ; variable name
6285
6286        LD   A,(length)
6287        CP   1
6288        CALL Z,prob
6289
6290 ; if 1 char. variable name then
6291 ; indicate a possible problem
6293
6294 sok:    LD   IX,368
6295 seloop: LD   E,(IX+0)
6296        LD   D,(IX+1)
6297        LD   L,(IX+2)
6298        LD   H,(IX+3)
6299        LD   (lineno),HL
6300        LD   A,L
6301        OR   H
6302        RET  Z
6303
6304 ; if line numnber=0 then finish
6305        CALL #BB1B
6306        RET  C
6309
6310 ; if key pressed, finished
6311
6312        CALL sline
6313
6314 ; scan the line
6315
6316        PUSH IX
6317        POP  HL
6318        DEC  DE
6319        ADD  HL,DE
6320        INC  HL
6321        PUSH HL
6322        POP  IX
6323
6324 ; get new line start into IX
6325
6326        JR   seloop
6327
6330 sline:  PUSH IX
6331        POP  HL
6332        PUSH IX
6333        PUSH DE
6334        POP  BC
6335        DEC  BC
6336        DEC  BC
6337        DEC  BC
6338        DEC  BC
6339        PUSH DE
6340        LD   DE,4
6341        ADD  HL,DE
6342
6343 ; length of text in BC and HL points
6344 ; at start of text line
6345
6346
6347 selop1: LD   DE,(stradd)
6348        PUSH BC
6349        LD   A,(DE)
6350        CP   (HL)
6351        JR   Z,hit
6352
6533 ; if character is same as first
6534 ; character of search string then
6535 ; jump off and check rest of string
6537
6538 notok:  POP  BC
6539        DEC  BC
6540        INC  HL
6541        LD   A,B
6542        OR   C
6543        JR   NZ,selop1
6544
6545 ; carry on and look at rest of string
6546
6549        POP  DE
```

```
6550        POP  IX
6551        RET
6552
6553 hit:   LD   A,(length)
6554        LD   B,A
6555 selop2: LD  A,(DE)
6556        CP   (HL)
6557        JR   NZ,notok
6558
6559 ; if mismatch, then go back
6560
6561        INC  HL
6562        INC  DE
6563        DJNZ selop2
6564
6565 ; get this far it's a find
6566
6567        CALL sefind
6568        JR   notok
6569
6570 sefind: CALL lret
6571        PUSH IX
6572        PUSH HL
6573        PUSH DE
6574        PUSH BC
6575        LD   HL,(lineno)
6576        CALL pnumhl
6577        POP  BC
6578        POP  DE
6579        POP  HL
6580        POP  IX
6581        RET
6582
6583 ; print the line number
6584
6585 prob:  LD   BC,22
6586        LD   HL,probm
6587        CALL messag
6588        RET
6589
6590 probm:  DEFM "Will also find tokens"
6591 stradd: DEFW 0000
6592 length: DEFB 00
6594
```

The |INK command is added by lines 5220-5370, and returns in BASIC variable the two colours associated with a given ink.

Lines 5450-6080 add the |FIND command, which returns the address of the first text byte of a given program line. The routine is on the whole self explanatory, but there are a couple of points of interest. Lines 5658 and 5659 clear the C flag. The Firmware routine called at &BB1B returns with C set if a key is pressed.

|SEARCH is the last routine to be added by this module. Again, the comments in the listing will explain most of the operation, but a quick explanation of the search method follows.

We start at the line length low byte of a line, and then save the line number for future reference. This is checked for 0, which would indicate the program end. In addition, the keyboard is examined for a key press. Either of these being found results in the routine finishing. Otherwise, the line is scanned. The line length is reduced by 4 to ensure that only the text part of the line is scanned, and the HL register is pointed to the first 'text' character in the program line (i.e. the first character after the line number high byte). Each byte of the line is then examined to see if it's the same as the first character of the search string, which is pointed to by DE. If a 'hit' is found, the routine saves the current HL position and scans the next 'n' characters, where n is the length of the search string. If a mismatch is found, the HL register is restored and the search continues. Otherwise, the line number is printed and the search continues.

And so on to Module 5, which completes this particular extension. The first routine, '|ROMREAD', allows you to examine the contents of ROM. The routine 'mode' sets the machine to mode 2, as 80 columns are required. The number of parameters are checked, and if all is well the two ROM's are enabled in lines 6608-6610. This means, of course, that this machine code must reside in an area of RAM that is not overlaid by ROM. If it were to be in such a place, the paging in of ROM would page OUT the actual machine code!

The 'eight' routine, at lines 6635 to 6697 prints out 8 bytes in hexadecimal on the left of the screen and the characters that these bytes represent on the right of the screen. The address of the first byte of the eight is also printed.

Lines 6618-6633 restore the ROMs to their original state, and then wait for a key to be pressed. (CALL &BB18). If the key wasn't ENTER, then the next 8 bytes are dumped. Otherwise, a RET is made to BASIC.

The 'phexa' routine at lines 6699-6739 simply prints out the A register contents as a 2 digit hexadecimal number.

```
6595 ; ******** MODULE 5 ********
6596
6597 ; this is the last of the modules
6598
6599 romr:   CALL mode
6600         CP  1
6601         JP  NZ,parerr
6602
6603         LD  L,(IX+0)
6604         LD  H,(IX+1)
6605
6606 RLOOP1: PUSH HL
6607         LD  (lineno),HL
6608         CALL #B906
6609         PUSH AF
6610         CALL #B900
6611
6612 ; enable upper and lower ROMs
6613
6614         CALL eight
6617         POP  AF
6618         CALL #B90C
6619         CALL #BB18
6620         POP  HL
6621
6622 ; wait for key press then restore HL
6623
6625         CP  13
6626         RET  Z
6628         LD  DE,8
6629         ADD  HL,DE
6633         JR  rloop1
6634
6635 eight:  LD  B,8
6636         LD  DE,buffer
6637         LD  HL,(lineno)
6638 dloop1: LD  A,(HL)
6639         LD  (DE),A
6640         INC  HL
6641         INC  DE
6642         DJNZ dloop1
6643
6644 ; get eight bytes from (HL)
6645
6646         LD  HL,(lineno)
6647         CALL pnumhl
6648         LD  A,10
6649         CALL #BB6F
6650
6651 ; print address of first byte then
6652 ; set column to 10
6653
6654         LD  B,8
6655         LD  DE,buffer
6656 dloop2: PUSH BC
6657         LD  A,(DE)
6658         PUSH DE
6659         CALL phexa
6660         POP  DE
6661
6662 ; get first byte and print in hex.
6663
6667         CALL #BB7B
```

```
6668        PUSH HL
6669
6670 ; get and save cursor position
6671
6672        LD  A,H
6673        ADD  A,30
6674        CALL #BB6F
6675
6676 ; prepare to print ascii codes
6677
6678        LD  A,(DE)
6679        PUSH DE
6680        CALL #BB5D
6681        POP  DE
6682        POP  HL
6683
6684 ; print codes and restore registers
6685
6686
6687        LD  A,H
6688        ADD  A,4
6689        CALL #BB6F
6690
6691 ; get ready to print next byte
6692
6693        INC  DE
6694        POP  BC
6695        DJNZ dloop2
6696        CALL lret
6697        RET
6698
6699 phexa:  PUSH BC
6700        LD  B,0
6701        LD  C,A
6702        RR  A
6703        RR  A
6704        RR  A
6705        RR  A
6706 prinlo: AND  #0F
6707        CP  10
6708        JR  NC,atof
6709        ADD  A,#30
6710        CALL #BB5A
6720        JR  phout
6730 atof:  ADD  A,#37
6731        CALL #BB5A
6732 phout: LD  A,B
6733        CP  1
6734        JR  Z,phfin
6735        LD  A,C
6736        LD  B,1
6737        JR  prinlo
6738 phfin: POP  BC
6739        RET
6740
6750 mode:  PUSH AF
6751        LD  A,2
6752        CALL #BC0E
6753        POP  AF
6754        RET
6755
6756
6757 dump:  CP  1
6758        JP  NZ,parerr
6763        CALL mode
6764        LD  L,(IX+0)
6765        LD  H,(IX+1)
6766 dump1: PUSH HL
6767        LD  (lineno),HL
6768        CALL eight
6769        CALL #BB18
6770        POP  HL
6771        CP  13
6772        RET  Z
6773        LD  DE,8
6774        ADD  HL,DE
6775        JR  dump1
6776
6777
6778 amplit: CP  2
6779        JP  NZ,parerr
6780        LD  L,(IX+0)
6781        LD  H,(IX+1)
6782        LD  (varadd),HL
6783
6784 ; get parameters
6785
6787        LD  A,(IX+2)
6788        CALL #BCC2
6789        JP  NC,serror
6790        LD  DE,(varadd)
6791
6792 ; get envelope number into A
6793 ; and get address of data into HL
6794
6796 aloop: LD  A,(HL)
6797        LD  (DE),A
6798        INC  DE
6799        INC  HL
6800        DEC  BC
6801        LD  A,B
6802        OR  C
6803        JR  NZ,aloop
6804        RET
6805
6806 ; transfer the bytes of data
6807
6810 tone:  CP  2
6811        JP  NZ,parerr
6812
6813 ; now same for tone envelopes
6814
```

```
6815        LD  L,(IX+0)
6816        LD  H,(IX+1)
6817        LD  (varadd),HL
6818        LD  A,(IX+2)
6819        CALL #BCC5
6820        JP  NC,serror
6821        LD  DE,(varadd)
6822 tloop: LD  A,(HL)
6823        LD  (DE),A
6824        INC HL
6825        INC DE
6826        DEC BC
6827        LD  A,B
6828        OR  C
6829        JR  NZ,tloop
6830        RET
6831
6832 serror: LD  BC,32
6833        LD  HL,smess
6834        CALL messag
6835        RET
6836
6837 smess:  DEFM "Can't find the envelope!"
```

|DUMP, coded in the lines 6757-6775 does the same job as |ROMREAD, but only dumps out the contents of RAM.

The final two commands, |AMPLITUDE (lines 6778-6807) and |TONE (lines 6810-6837) allow the currently in use envelopes to be examined. Two firmware routines are used to get the address of either the Amplitude or Tone envelopes, and the data held at these locations is transferred to a user defined space.

Notes on the Assembler Used.

The listings were made under the Amsoft 'DEVPAC' Assembler. '#' is used to prefix a hexadecimal number, rather than the more usual '&'. In addition, 'ORG' simply specifies the address at which you wish the resultant machine code to be placed. Each label is followed by a ':'. The DEFS n directive simply reserves an area of 'n' bytes. DEFM a$ puts the string 'a$' into memory at that point, DEFW nn sets two bytes to the value 'nn' and DEFB sets a single byte to hold the value 'n'.

Using the Commands

|RESET. This sets screen colours, keyboard, sound etc. back to their

original state. Useful if you've been experimenting with colour combinations and come up with something unreadable.

|PAUSE,n. This generates a time delay of 'n' 50ths of a second. It can also be terminated by pressing a key.

|GET,@a%. 'a%' can be any integer variable. The command waits until a key is pressed and returns the ASCII code of the character in the variable.

|CRTC,register,value. Writes 'value' to register 'register' of the 6845 Cathode Ray Tube Controller chip. We'll see this in use in the next Chapter.

|SOUND,register,value. Writes 'value' to register 'register' of the Programmable Sound Generator. This command allows you to access the PSG directly from BASIC.

|BRKOFF turns off the normal ESCAPE or BREAK key action from within a running program.

|BRKON turns the normal Escape key action back on again.

|CAT. This command performs an enhanced CAT function by providing type, length and load address information for files.

|MOTOR,n. It is an annoying feature of the 464 tape system that you have to CAT the tape to actually find a blank bit of tape. This means you can't do anything else while this is happening. Where 'n'=1 the tape drive is turned on, and pressing play will allow the tape to be heard without having to CAT. Where 'n'=0 the drive is turned off.

|SYMBOL,char,address. This command returns to you the 8 byte pattern definition of the character whose ASCII code is 'char'. The 8 bytes are returned to the 8 memory locations starting at 'address'.

|INK,colour,@a%. This returns the two colours assigned to INK 'colour'. The two colours are returned in the variable 'a%'. INT(a%/256) will give the first colour and a% MOD 256 will give the second colour.

|TONE,n,address. This command will return the bytes of the Tone envelope 'n' to you. They will be stored at address 'address'.

|AMPLITUDE,n,address. As for |TONE, but returns an Amplitude Envelope.

|FIND,n,@a%. This returns the address, in 'a%', of the first byte after the line number of line 'n'. Should the line not exist, you can exit the command by pressing a key.

|SEARCH,n$,flag%. This will search the program text for occurrences of the text string 'n$'. If 'flag%' is set to 1 then variable names will be searched for. If set to 0, then simply text will be searched for. The numbers of lines in which the text is found are printed to the screen. There are a couple of points to note when using this command. The first is that searching for single character variable names will occasionally throw up lines which do not appear to contain that variable. However, they do contain a token which has the value of the encoded single letter variable name. In true computing fashion, we'll turn this bug into a feature by documenting it, and say that by looking at the Token tables in the Appendices, and by subtracting 128 from the token of interest and putting the resultant character in as n$, we can search for Tokens as well as text.

The second point is that a search for a string, like 'cat' will also find strings of which 'cat' is part, such as 'scatter' or 'catapult'.

|LLIST,n. This command offers the option of directing any material sent down Stream 8 to the printer to both printer AND screen. |LLIST,1 will enable this and |LLIST,0 will return things to normal. Thus if, like me, your printer is some distance away from the computer you can see what is being printed out without wandering between printer and computer. An idle programmers command, this one!

|ROMREAD,address. This command allows you to inspect the contents of ROM. 'address' holds the address of the first byte you want displaying, and then lines of 8 bytes will be displayed. To exit the routine, press ENTER. To see another 8 bytes, press SPACE. The bytes are displayed in hexadecimal, and the characters represented by these bytes are also displayed.

|DUMP,address. This is the same as |ROMREAD, but allows you to inspect RAM instead of ROM.

A dump of the bytes needed for this program will be given at the end of this Chapter.

Machine Code and BASIC

The rest of this Chapter will be devoted to the ways in which machine code and BASIC programs can interact with each other in ways other than the simple passing of variables. Some of the routines here will make use of calls into the Amstrad BASIC ROM, and so are rather 'dirty'. The work was done on an Amstrad 464 with BASIC Version 1.0. So, if you haven't got this version of BASIC, be warned; the routines may not work as they stand. However, notes are given in the Appendices on altering the routines to run on BASIC 1.1.

Patching Jumpblocks

In a computer, the Operating System deals with the tedious jobs, such as putting a character on the screen, reading the keyboard, interacting with discs or cassette, etc. In the Amstrad, the OS routines are accessed by calls to addresses in the central RAM area which are called Jumpblocks. These contain a three byte instruction which simultaneously pages in the appropriate ROM and jumps to the required address within that ROM. The reason why we go through these 'middle men' is simple; if later versions of the OS are written, only the contents of the Jumpblocks need be altered to maintain compatibility with earlier versions. Thus, if programmers have kept to the rules and used the Jumpblock to access OS routines, their programs will work on any version of the Operating System.

There is also a second advantage; we can alter the contents of Jumpblocks ourselves to point to our own code, and so alter the behaviour of Operating System routines. In fact, the |LLIST command above was an example of this.

Now, as BASIC uses the Jumpblock entries itself, we can, by patching the Jumpblocks in this way, also alter the behaviour of BASIC to a small degree. In this section we'll see how this can be done; don't worry, it's a surprisingly easy and painless process!

How it's done

1. The current values stored in the Jumpblock of interest should be stored somewhere safe, so that a CALL or JP can be made to old Jumpblock contents if required.

2. The Jumpblock should now be patched with a JP address instruction to point to your new code. The first byte holds the code '&C3', which is the op-code for JP, and the other two bytes hold the address of the routine stored low byte first. It's good practice to ensure that code does exist at 'address' before the jumpblock patch is put in. If there isn't, and the OS calls the Jumpblock, the consequences could be nasty!

3. If your code replaces the normal OS routines completely, then a RET can be used to finish the new routine. This isn't, however, common; patches in to the OS usually only partially replace the normal OS behaviour. In this case, there are two options.

> i/. CALL the usual Jumpblock entry, then do a RET. This was done in the |LLIST command.
>
> ii/. JP to the usual Jumpblock entry. This will result in the normal OS routine being executed as a final act before control returns to the routine which originally called the Jumpblock address.

One point that is obvious is that we don't really need to know what the Jumpblock entries are; we simply copy them and use them as required.

One final point to note is that of interrupts. If a Jumpblock were to be altered just as an interrupt was to call the routine, then a rather nasty crash would probably result. So, to get around this it's useful to disable the Z-80 interrupts before altering the Jumpblock contents and re-enable them after setting the Jumpblock up. Of course, this needn't be done for all such operations.

SPLITON and SPLITOFF

To demonstrate the addition of another command that alters a Jumpblock, examine the SPLITON/SPLITOFF listing. This adds a useful facility for putting each statement in a multistatement line on a separate line when a listing is made, thus making the listing easier to read. It works by intercepting the bytes sent to &BB5A and checking for the ASCII code for ':'.

In addition, the routine recognises if a colon appears in quotation marks. In this case it doesn't split the line up. In the program, lines 60-160 set the commands up and make a copy of the Jumpblock that we'll be altering. The 'spon' routine, at lines 180-220, turns on the line splitting facility by putting a:

```
JP      newcode
```

instruction in to it. Using JP instructions like this means that the additional code must be in the central area of RAM – i.e. that area of RAM between &4000 and &BFFF which is not overlaid by RAM.

'spoff' turns off the line splitting facility by restoring the usual contents to the &BB5A Jumpblock.

```
10 ; An RSX routine to add the
20 ; |SPLITON and |SPLITOFF
30 ; commands
40
50       ORG 37800
60       LD  HL,worksp
70       LD  BC,table
80       CALL #BCD1
90       LD  A,(#BB5A)
100      LD  (add1),A
110      LD  A,(#BB5B)
120      LD  (add2),A
130      LD  A,(#BB5C)
140      LD  (add3),A
150      RET
160
170 ; save the normal jump block
180 ; contents and set up the
190 ; command tables
200
210 table: DEFW names
220      JP  spon
230      JP  spoff
240 names: DEFM "SPLITO"
250      DEFB "N"+128
260      DEFM "SPLITOF"
270      DEFB "F"+128
280      DEFB 00
290
300 spon:  LD  A,#C3
310      LD  HL,newcod
320      LD  (#BB5A),A
330      LD  (#BB5B),HL
340      RET
350
360 ; put the address of the routine
370 ; into the jumpblock and a jump ins.
380
390
400 spoff: LD  A,(add1)
410      LD  (#BB5A),A
420      LD  HL,(add2)
430      LD  (#BB5B),HL
440      RET
450
460 newcod: CP  ":"
470      JR  NZ,dont
480      PUSH AF
490      LD  A,(flag)
500      CP  0
510      JR  Z,lret
520      POP AF
530      JP  add1
540
550 ; check to see if it's a colon
560 ; and also make sure that it's not
570 ; within quotation marks
580
```

```
590
600 lret:  LD   A,13
610        CALL add1
620        LD   A,10
630        CALL add1
640        CALL space
650        POP  AF
660        LD   A,":"
670        JP   add1
680
690 ; lret prints out a carriage return
700 ; and some spaces if the colon is a
710 ; statement separator
720
730
740 dont:  CP   34
750        JR   NZ,out
760        PUSH AF
770        LD   A,(flag)
780        CP   0
790        JR   Z,setflg
800        LD   A,0
810        LD   (flag),A
820        JR   out1
830 setflg: LD  A,1
840        LD   (flag),A
850 out1:  POP  AF
860 out:   JP   add1
870
880 ; dont checks for quotes and if it's
890 ; a closing quote then set flag to
900 ; 0. If it's an opening quote set the flag
910 ; to 1.
920
930 space: LD   B,5
940 loop:  LD   A,32
950        CALL add1
960        DJNZ loop
970        RET
980 flag:  DEFB 00
990 add1:  DEFB 00
995 add2:  DEFB 00
996 add3:  DEFB 00
```

The code to which we point the Jumpblock, 'newcod' (Lines 300-580) first sees if the character is a colon. If it isn't, then a jump is made to 'dont' where the character is checked to see if it's a quotation mark. If it is, then we ascertain whether it's an opening or closing quote depending upon the value in 'flag'. The character is then printed, by making a JP to the old Jumpblock contents, thus terminating the routine.

If the character isn't a colon or a quote, then it's simply printed. For colons, a check is made on 'flag' to see if we're in the middle of two quotation marks. If we are, it's printed as usual. If it is a statement separator, then a carriage return and 5 spaces are printed, followed by the colon which is printed by a jump back to the old contents of &BB5A.

Other Jumpblocks could easily be altered; for example, it should be possible to alter the Jumpblocks for Cassette routines to point to user defined tape handling routines, new printer routines and so on. Of course, there's little point in altering some Jumpblocks, but the above techniques are applicable to any such operation.

Errors in Machine Code Programs

One thing you'll have probably noticed is that the messages printed out in response to error conditions in the RSX commands above are not real 'Error Messages' as are produced by BASIC errors such as 'Syntax Error', etc. The messages generated in the RSX routines won't stop a running program at all; in this section I'll explain how we

can get an error generated in a machine code routine to be the same as an Error generated in BASIC. That is, ERR and ERL will be set up, and the program will stop with the appropriate message. If an ON ERROR GOTO nn instruction has been issued from BASIC, then it will be responded to in the usual fashion. The only drawback is that we will be limited to the Error Messages and Numbers that exist in the BASIC ROM. However, this isn't too bad, and it does allow us to trap errors within machine code routines that would otherwise be difficult to deal with. Once such an error has been generated, action can be taken in an ON ERROR trap to clear up the error condition.

A demonstration machine code routine is shown in the 'ERROR' listing. This could, of course, be put in any of your programs. Line 110 pages in the Upper ROM; this is essential, as otherwise we could call a screen RAM location rather than a ROM routine! The actual business of generating an error is simple; the A register is set up with the number of the Error that you want to generate and then a call to the ROM routine at &CA93. This is, incidentally, an entry in to the interpreter code that takes care of the 'ERROR' command. We are effectively, therefore, performing a machine code version of the ERROR command. To see this routine in action, simply call it and you'll see the error generated.

```
 10 ; demo program to generate errors
 20 ; from within machine code that
 30 ; are picked up by the usual BASIC
 40 ; error handler
 50
 60 ; the address given is for BASIC
 70 ; version 1.0
 80
100      ORG  38000
110      CALL #B900
120
130 ; enable the BASIC ROM so we
140 ; can call the routine
150
160      LD  A,4
170
180 ; put error number in A
190
200      CALL #CA93
210
220 ; call the routine
```

Thus, this routine offers us the chance to generate error messages from within machine code routines and also allow machine code routines to direct the flow of a BASIC program by use of ON ERROR GOTO.

Remember that the address given in the listing is for BASIC 1.0.

The ROM routine is called, but the return is made via the BASIC Interpreter. Examination of the ROM code around this point indicates that the Z80 Stack Pointer is reset to &C000, which thus clears the stack. This could be the reason why there is no ON ERROR GOSUB instruction.

There are other ways in which machine code routines can affect a BASIC program other than by the ON ERROR GOTO instruction. One way is to use Event Blocks.

Use of Event Blocks

I will not go in to the details of how Events are implemented on the Amstrad computer; this is adequately covered in other places, such as the Amsoft 'Concise Firmware Manual'. In this section I will examine how use of Event blocks can help the BASIC programmer. Of course, this will involve some machine code programming.

The simplest Event to use is the Break Event, usually generated at the keyboard when ESCAPE is pressed. As you will be aware, pressing ESC twice in a running program will cause the program to be terminated, or, if an ON BREAK GOSUB has been issued, cause execution of the relevant subroutine. It is not too difficult to generate this Event from within a machine code routine, and once called will, within a running program, simulate the action of ESC being pressed twice. The machine code routine to do this is shown in listing 'BREAK'. It uses the technique just outlined to change the contents of a jumpblock.

First of all, a few words about how the Break Event is usually processed. A single press of the Escape key leads to the BASIC Break Handling routine at address &C45E (Version 1.0) being entered. This leads to a suspension of any sounds being generated, and a halt to the execution of the program by a call to MC WAIT CHAR.

```
10      ORG 38000
20
30 ; first of all, save the contents
40 ; of the MC WAIT CHAR jumpblock
50 ; entry into 'temp'
60
70      LD  A,(#BB06)
80      LD  HL,(#BB07)
90      LD  (temp1),A
100     LD  (temp2),HL
110
120 ; now set the jumpblock up to point
130 ; to our new routine
140
150     LD  HL,newr
160     LD  (#BB07),HL
170     LD  A,#C3
180     LD  (#BB06),A
190
200 ; now we arm the break events
210
220     LD  DE,#C45E
230     LD  C,#FD
240     CALL #BB45
250
260 ; kick the break event
270
280     CALL #BB4B
```

```
290       RET
300
310
320 newr:  LD   A,(temp1)
330        LD   (#BB06),A
340        LD   HL,(temp2)
350        LD   (#BB07),HL
360
370 ; above code resets the jumpblock
380
390        LD   A,#FC
400        SCF
410        RET
420
430 ; finally simulate pressing escape
440
450 temp1:  DEFB 00
460 temp2:  DEFW 0000
```

This causes the machine to wait until a key is pressed; should this return with the number &FC in the A register, then the Escape key has been pressed and it is clear that the user wishes a full Break; the OS then generates the full Break Condition. Any other number in the A register will lead to the Break event being discarded and the program execution carrying on. This program works by altering MC WAIT CHAR to point to a routine of our own just before the Break Event is kicked. Thus, when the ROM routine calls MC WAIT CHAR it actually calls our routine, which returns the code &FC in A. In addition, our new MC WAIT CHAR routine resets the Jumpblock for MC WAIT CHAR to its original contents. The purpose of this messing about with MC WAIT CHAR is to simulate the second pressing of the Escape Key, thus completing the Break Event. CALLing this routine from a running program will cause any ON BREAK GOSUB routine that is in operation to be entered.

Lines 70-100 of BREAK save the usual contents of the MC WAIT CHAR Jumpblock into three temporary locations. Lines 150-180 set up the Jumpblock with a JP instruction to pass control to 'newr' when the routine is called from this point on. Lines 220-240 arm the Break Event with the address of the Event Routine in DE and the ROM selection code in the C register. It is necessary to arm these events because the default state of the Break Event is disarmed. Finally, we kick the Break Event in line 280, thus causing the routine at &C45E to be entered.

The 'newr' routine at line 320 is entered only once, when the &C45E routine calls MC WAIT CHAR to get the second press of the Escape Key. So, the first job to do is to set up MC WAIT CHAR with it s usual contents. This is done in lines 320-350.

Lines 390-400 put &FC in the A register, thus simulating an Escape Key press, and sets the carry flag, thus simulating the normal return conditions to be expected after a call to MC WAIT CHAR.

Finally, a RET is made to finish our routine.

The Break Event thus generated offers you another means of altering the running of a BASIC program from within machine code, thus allowing the various ON BREAK instructions from BASIC to be interacted with by machine code.

The Frame Fly Event

A more typical event is the frame fly event, a timed interrupt that occurs 50 times per second on UK machines. It is generated by the Video Hardware after each frame of the display has been completed. It is thus a means by which we can carry out a given machine code routine very regularly. The routine 'KEY' shows this in action, reading the keyboard 50 times per second and putting the ASCII code of any key pressed into an integer variable. This variable thus

```
10      ORG 38000
20      LD  L,(IX)
30      LD  H,(IX+1)
40      LD  (varadd),HL
41
42 ; the above lines get the address of
43 ; the variable passed over as a
44 ; parameter and is then stored
45
50      LD  HL,blk
60      LD  B,5
70      LD  DE,routin
80      LD  C,#FD
90      CALL #BCEF
91
92 ; then set up an event block for
93 ; a synchronous event with priority
94 ; 4. This gives a class in B of 5
98
100     LD  HL,spare
110     CALL #BCDA
120     RET
121
122 ; now add the event block to the list
123
140 routin: CALL #BB09
150         JR  NC,out
160         CALL putin
170         RET
180 out:    LD  A,0
190         CALL putin
200         RET
210
211 ; this 'routin' reads the keyboard every
212 ; 1/50th of a second and if a key is
213 ; pressed the value returned is put in
214 ; a variable
215
220 putin:  LD  HL,(varadd)
230         LD  (HL),A
240         INC HL
250         LD  (HL),0
260         RET
270
280 spare:  DEFW 0000
290 blk:    DEFS 10
300 varadd: DEFW 0000
```

reflects the status of the keyboard at any time in a running program. The routine is activated by a line such as:

```
10  char%=0:CALL 38000,@char%
```

From this point on, char% will hold the value of a key being pressed at that instant in time; it will be, at most, 1/50 th second out of date. The advantage of this over the usual INKEY operation is that the act of reading the keyboard is now slightly faster because all that we need to do is to access the variable!

The listing 'KEY' shows this routine. Lines 20-40 transfer the address of the integer variable passed as a parameter of the CALL to a temporary storage location; the variable address is pointed to by the IX register.

Lines 50-90 set up an Event Block. HL is pointed to workspace, DE is given the address of the Event Routine and B is the class of the

Event. This is described in the Firmware manual, and all I will say here is that we've set it up as a Synchronous Event, priority 4, near address. The priority is self explanatory, and the term 'near address' simply indicates that the routine pointed to by DE is in central RAM. &BCEF is the jumpblock routine to set it up. The next step is to add this Event Block to the list of Events that are under the control of the Frame Fly. This is done by loading HL with the address of the Event Block minus two, and then calling &BCDA. This is done in lines 100-120.

That completes the routine as far as the setting up of the Event goes. Each time the Event Block is 'kicked' by the Frame Fly Interrupt, the machine code at 'routin' will be executed as soon as possible after the kick. This simply reads the keyboard (Line 140) and if no key is pressed puts zero in the variable (Lines 180-200). If a key is pressed, then the ASCII code is put in the variable. (Line 160).

The routine should only be added to the Frame Fly list once, and once active will be until you reset the machine or use the DEL FRAME FLY routine in the OS to remove it from the Frame Fly list.

ON KEY GOSUB

Let's now look at a useful RSX command for games programmers; a routine that causes a subroutine to be executed whenever a key is pressed. This routine uses the BREAK program and the KEY program, and is used as follows.

After the |KEYON,@char% instruction, pressing a key will cause the a Break Event to occur. This can then be detected by an ON BREAK GOSUB subroutine. The value of the key that caused the routine to be entered will then be in 'char%'; examination of this will also indicate if the routine was entered by a normal Break Event rather than a key press. |KEYOFF turns off this behaviour, and should be used BEFORE |KEYON is used.

```
10 ; RSX to add |KEYON and |KEYOFF
20 ; commands to Amstrad BASIC
30
40       ORG  37600
50       JP   doit
60
70 ; jump around the tables
80
90 name:  DEFM "KEYNO"
100      DEFB "N"+128
110      DEFM "KEYOF"
120      DEFB "F"+128
130      DEFB 0
140 table: DEFW name
150      JP   keyon
160      JP   keyoff
170 doit:  LD   HL,worksp
180        LD   BC,table
190        CALL #BCD1
200        RET
210
220 ; set up the two RSX commands
230
240 keyon: CP   1
250        RET  NZ
260
270 ; check for 1 parameter
280
290        LD   A,(flg)
300        CP   1
310        RET  Z
320
```

```
330 ; if keyon is already active we
340 ; simply return
350
370      LD  A,1
380      LD  (flg),A
390
400 ; set the flag to indicate it's active
410
420
430      LD  L,(IX)
440      LD  H,(IX+1)
450      LD  (varadd),HL
460      LD  HL,blk
470      LD  DE,frame
480      LD  B,5
490      LD  C,#FD
500      CALL #BCEF
510      LD  HL,spare
520      CALL #BCDA
530      RET
540
550 ; set up frame fly block to
560 ; read the keyboard every 1/50th
570 ; of a second using the code at frame
580
590 worksp: DEFS 4
600 varadd: DEFW 0000
610 spare:  DEFW 0000
620 blk:    DEFS 10
630 flg:    DEFB 0
640
650 keyoff: CP   0
660      RET NZ
670      LD  A,0
680      LD  (flg),A
690      LD  HL,spare
700      CALL #BCDD
710      LD  HL,(varadd)
720      LD  (HL),0
730      INC HL
740      LD  (HL),0
750      RET
760
770 ; turns off the frame fly routine
780 ; and resets the flag byte and also
790 ; zeros the variable used.
800
830 frame:  CALL #BB09
840      JR  NC,out
850      CALL putin
860      RET
870 out:  LD  A,0
880      LD  HL,(varadd)
890      LD  (HL),0
900      INC HL
910      LD  (HL),0
920      RET
930 putin: LD  HL,(varadd)
940      LD  (HL),A
950      INC HL
960      LD  (HL),0
970      LD  A,(#BB06)
980      LD  HL,(#BB07)
990      LD  (temp1),A
1000     LD  (temp2),HL
1010     LD  HL,newr
1020     LD  (#BB07),HL
1030     LD  A,#C3
1040     LD  (#BB06),A
1050     LD  DE,#C45E
1060     LD  C,#FD
1070     CALL #BB45
1080     CALL #BB4B
1090     RET
1100
1110 ; the frame routine reads the keyboar
1120 ; and if a keypress is detected then
1130 ; a break event is generated
1170
1180 temp1: DEFB 0
1190 temp2: DEFW 0000
1200
1210 newr:  LD  A,(temp1)
1220     LD  (#BB06),A
1230     LD  HL,(temp2)
1240     LD  (#BB07),HL
1250     LD  A,#FC
1260     SCF
1270     RET
```

The subroutine can then carry out certain actions depending upon the key pressed, such as modifying the position of a character on the screen, and so on.The KEYON listing shows the routine, and the Hex for the routine is given at the end of the Chapter. The Hex is for address 37600.

Lines 50-200 set up the RSX tables. Lines 240-630 and lines 830-1270 deal with the KEYON command. The main parts of these routines have already been dealt with; the code at lines 280-310

simply ensures that the Frame Fly Event Block is only added to the Frame Fly list once. The KEYOFF code, between lines 650-750, takes this Frame Fly Event Block off of the Frame Fly list, and sets the flag byte and variable accordingly. The lines of BASIC below show how the routine might be used:

```
    1 CALL 37600: REM set up RSX
    2  char%=0:|KEYOFF:|KEYON,@char%
    3 ON BREAK GOSUB 10000
    4
    5 REM rest of program
    6
10000 REM Break Routine
10010 IF char%=252 OR char%=0 THEN |KEYOFF:END
10020 PRINT char%
10030 RETURN
```

Line 10010 of the above program trap the normal pressing of the Escape key, turning off the ON KEY GOSUB event and stopping the program. Line 10020 just prints the ASCII code of the key that was pressed. Line 1 sets up the RSX commands, and line 2 sets the variable to zero and initialises the |KEYON command to put the ASCII code in the 'char%' variable. Note how KEYOFF is used to set the system in a 'known' state before KEYON is used.

There are other ways in which machine code routines can interact with BASIC; there will be other occasions in this book when such routines will be used. However, we've now completed our RSX routines. As to entering the routines, the above assembler listings can be typed in if you've got an Assembler. Otherwise, use the BASIC loader program to enter the Hexadecimal listings given on pages 176 to 178. To enter the listings, start at the top left of the page and work across the page from left to right, top to bottom. The program is reasonably efficient and will allow you to enter small machine code programs.

```
 10 REM machine code editor program
 20 REM to allow bytes to be entered
 30 REM into memory
 40 REM Joe Pritchard
 50 REM September 1985
 60 :
 70 DIM array%(10,10)
 80 in$="0123456789ABCDEFabcdef#"+CHR$(13)
 90 GOSUB 1000
100 GOTO 90
110 :
```

```
1000 MODE 1
1010 RESTORE
1020 LOCATE 10,1
1030 PEN 3:PRINT"Options Available"
1035 j=1:FOR i=4 TO 7:READ a$:LOCATE 12,i*2:PEN 3:PRINT j;".
     ";:PEN 1:PRINT a$:j=j+1:NEXT
1040 PEN 3:LOCATE 10,18:PRINT "Press 1,2,3 or 4"
1050 a$=INKEY$:IF a$="" THEN GOTO 1050
1060 IF INSTR("1234",a$)=0 THEN LOCATE 1,1:PRINT
     CHR$(7):GOTO 1050
1070 ON INSTR("1234",a$) GOSUB 2000,3000,4000,5000
1080 RETURN
1081 :
1082 :
2000 REM enter bytes subroutine
2010 MODE 2
2015 WINDOW#1,1,80,1,23
2020 WINDOW#7,1,80,24,25:INPUT#7,;"What is the start address:
     ",add
2021 finflg%=0:counter%=1
2030 PRINT#1,add;"     ";
2040 GOSUB 8000
2041 counter%=counter%+1
2044 POKE add,num%:add=add+1
2045 IF counter%=11 THEN counter%=1:PRINT#1:GOTO 2030
2046 IF finflg%=1 THEN RETURN
2050 GOTO 2040
2060 :
3000 REM subroutine to edit bytes in memory
3002 MODE 2
3004 WINDOW#1,1,80,1,23
3006 WINDOW#7,1,80,24,25:INPUT#7,;"What is the start address:
     ",add
3008 finflg%=0:counter%=0:initadd=add
3010 PRINT#1,add;"     ";
3020 FOR i=0 TO 9:PRINT#1,HEX$(PEEK(add+i),2);"
     ";:array%(i,counter%/10)=PEEK(add+i):NEXT
3030 PRINT#1:add=add+10
3040 counter%=counter%+10
3050 IF counter%<>110 THEN GOTO 3010
3060 :
3070 REM now the screen editor
3080 xp=13:yp=1:xa=0:ya=0:finflg%=0
3090 a$=INKEY$:IF a$="" THEN GOTO 3090
3100 char%=ASC(a$):k%=0
3105 IF a$=" " THEN k%=1:finflg%=1
3110 IF char%=240 AND yp>1 THEN ya=ya-1:yp=yp-1:k%=1
3120 IF char%=241 AND yp<11 THEN ya=ya+1:yp=yp+1:k%=1
3130 IF char%=243 AND xp<58 THEN xp=xp+5:xa=xa+1:k%=1
3140 IF char%=242 AND xp>13 THEN xp=xp-5:xa=xa-1:k%=1
3150 IF k%=0 THEN LOCATE 1,1:PRINT CHR$(7):GOTO 3090
3160 IF finflg%=1 THEN RETURN
3170 LOCATE #1,xp,yp:GOSUB 9000:IF finflg%=1 THEN RETURN
3180 GOTO 3090
3190 :
4000 REM save bytes routine
```

```
4010 CLS:LOCATE 10,1:PEN 3
4020 PRINT "Save bytes to Tape"
4030 PEN 1
4040 PRINT:PRINT:PRINT:PRINT
4050 PRINT "Position tape, press a key when ready"
4055 n$=""
4060 a$=INKEY$:IF a$="" THEN GOTO 4060
4070 PEN 3:PRINT:PRINT:PRINT"Start Address     ";:PEN
     1:INPUT"",add
4080 PEN 3:PRINT:PRINT:PRINT"End Address       ";:PEN
     1:INPUT"",fin
4090 PEN 3:PRINT:PRINT:PRINT"File Name         ";:PEN
     1:INPUT"",n$
4099 n$="!"+n$
4100 SAVE n$,B,add,(fin-add)
4110 RETURN
4120 :
5000 REM load bytes routine
5010 CLS:LOCATE 10,1:PEN 3
5020 PRINT "Load bytes from Tape"
5030 PEN 1
5040 PRINT:PRINT:PRINT:PRINT
5050 PRINT "Position tape, press a key when ready"
5060 a$=INKEY$:IF a$="" THEN GOTO 5060
5070 PEN 3:PRINT:PRINT:PRINT"File Name         ";:PEN
     1:INPUT"",n$
5100 PEN 3:PRINT:PRINT:PRINT"Load Address      ";:PEN
     1:INPUT"",add
5110 n$="!"+n$
5120 LOAD n$,add
5130 RETURN
5140 :
8000 REM routine to get two hex characters
8001 num$=""
8005 WINDOW#2,(POS(#1)+1),(POS(#1)+2),VPOS(#1),VPOS(#1)
8010 a$=INKEY$:IF a$="" THEN GOTO 8010
8020 IF INSTR(in$,a$)=0 THEN LOCATE 1,1:PRINT CHR$(7);:GOTO
     8010
8025 IF (a$=CHR$(13) AND num$<>"") THEN SOUND
     1,200,20:num%=VAL("&"+RIGHT$(num$,2)):LOCATE
     #1,POS(#1)+4,VPOS(#1):RETURN
8026 IF a$="#" THEN finflg%=1:RETURN
8030 IF a$<>"#" AND a$<>CHR$(13) THEN num$=num$+a$:PRINT
     #2,a$;
8040 GOTO 8010
8050 :
9000 WINDOW#2,(POS(#1)+1),(POS(#1)+
     3),VPOS(#1),VPOS(#1):CLS#2
9010 a$=INKEY$:IF a$="" THEN GOTO 9010
9015 char%=ASC(a$):IF char%<240 OR char%>243 THEN
     num$="":GOSUB 8020:POKE (initadd+10*ya+
     xa),num%:array%(xa,ya)=num%:RETURN
9020 PRINT#2,HEX$(array%(xa,ya),2)
9030 RETURN
10000 DATA "Enter Bytes","Edit Bytes","Save Bytes","Load Bytes"
```

Using the Machine Code Loader

On running the program you will be confronted by a 4 option menu. To enter a given option, simply press the relevant number on the keyboard.

Option 1. Enter Bytes

The start address should be typed in in decimal or hexadecimal. This will then be printed in the top left of the screen. The bytes can then be typed in as two digit hexadecimal numbers followed by the <ENTER> key.

If you enter an incorrect number, simply type in the correct one; only the last two digits entered will be treated as the Hex number when the <ENTER> key is pressed. If you type in too many numbers at this point, (more than 255), an error may be generated. If this happens simply re-run the program and carry on with the Edit or Enter Bytes option. On pressing <ENTER> a 'beep' will be generated and you will be able to enter the next byte. After 10 bytes the current address will be printed at the left of the screen and you will be able to enter the next ten bytes. When you've entered all the bytes, simply press '#'. This will return you to the main menu.

Option 2. Edit Bytes

The start address is entered and then the next 110 bytes will be printed to the screen. The Cursor keys can then be used to move 'around' this grid of bytes. The cursor keys will allow a 'space' to be positioned over any of the bytes displayed. Once the space is displayed, pressing the cursor keys will remove the space and restore the byte thus covered to it s original value. Pressing the cursor key a second time will move the 'space' to another adjacent byte. Alternatively, when the 'space' is displayed a two digit hexadecimal number can be entered as in the 'Enter Bytes' option. This will be entered at the address of the byte that was originally covered by the 'space'.

This allows you to edit the contents of memory. The best way to get used to this editor is to simply use it!

Option 3. Save Bytes

This option simply allows you to save a block of memory to tape. Simply position the tape, enter the start address, which is the address of the first byte to be saved, the finish address, which is the address of the last byte, and the name under which you wish the bytes to be saved. I tend to always add a few bytes to the finish address so as to ensure that I get all the bytes I want on to tape.

Option 4. Load Bytes

This simply allows you to load bytes in to memory. Simply enter the name of the tape file and the address to which the bytes are to be loaded.

You might like to add more facilities to the program, which, as it stands, is very basic (pardon the pun!!). But it will allow the routines listed in this chapter, and any others in the book, to be entered. Keep it handy, we'll be looking at memory in Chapter 3 using this program. This chapter is now finished by the byte listings for the various machine code routines. Good Luck with them!

HEX DUMPS FOR RSX PROGRAMS

START ADDRESS 38000 RSX

```
21  92  94  01  96  94  CD  D1  BC  3A  EE  BD
32  28  95  3A  F1  BD  32  29  95  3A  F2  BD
32  2A  95  3A  F3  BD  32  2B  95  C9  92  94
96  94  CE  94  C3  93  95  C3  AF  95  C3  E0
97  C3  9F  99  C3  33  98  C3  CC  95  C3  FC
98  C3  93  97  C3  5A  95  C3  70  95  C3  CB
97  C3  C1  99  C3  E6  99  C3  6B  97  C3  54
95  C3  4D  95  C3  25  96  C3  7F  95  52  45
53  45  D4  50  41  55  53  C5  46  49  4E  C4
44  55  4D  D0  53  45  41  52  43  C8  47  45
D4  52  4F  4D  52  45  41  C4  4C  4C  49  53
D4  43  52  54  C3  53  4F  55  4E  C4  49  4E
CB  41  4D  50  4C  49  54  55  44  C5  54  4F
4E  C5  53  59  4D  42  4F  CC  42  52  4B  4F
46  C6  42  52  4B  4F  CE  43  41  D4  4D  4F
54  4F  D2  00  C3  C3  F8  07  21  39  95  7E
FE  00  C8  CD  5A  BB  23  18  F6  50  61  72
61  6D  65  74  65  72  20  45  72  72  6F  72
21  21  0D  0A  00  3A  28  95  32  EE  BD  C9
3E  C9  32  EE  BD  C9  FE  02  C2  2C  95  DD
7E  02  01  00  BC  ED  79  DD  7E  00  01  00
BD  ED  79  C9  FE  02  C2  2C  95  DD  7E  02
DD  4E  00  CD  34  BD  C9  FE  01  C2  2C  95
DD  7E  00  FE  00  28  04  CD  6E  BC  C9  CD
71  BC  C9  CD  4E  BB  CD  00  BB  CD  BA  BB
CD  02  BC  CD  65  BC  CD  A7  BC  CD  37  BD
3A  EE  BD  32  28  95  C9  FE  01  C2  2C  95
DD  4E  00  DD  46  01  C5  CD  19  BD  CD  1B
BB  38  07  C1  0B  78  B1  20  F1  C9  C1  C9
FE  01  C2  2C  95  DD  6E  00  DD  66  01  CD
18  BB  77  23  36  00  C9  C5  7E  CD  5A  BB
C1  23  0B  78  B1  20  F4  C9  3E  0D  CD  5A
BB  3E  0A  CD  5A  BB  C9  11  10  27  CD  12
96  11  E8  03  CD  12  96  11  64  00  CD  12
96  11  0A  00  CD  12  96  11  01  00  AF  37
3F  ED  52  38  03  3C  18  F7  19  C6  30  E5
CD  5A  BB  E1  C9  3E  2C  11  40  00  21  25
97  CD  A1  BC  38  01  C9  21  25  97  01  10
00  CD  DF  95  CD  EC  95  21  EE  96  01  09
00  CD  DF  95  DD  21  25  97  DD  6E  18  DD
66  19  CD  F7  95  CD  EC  95  21  F7  96  01
09  00  CD  DF  95  DD  6E  15  DD  66  16  CD
F7  95  CD  EC  95  21  00  97  01  09  00  CD
DF  95  DD  6E  10  AF  67  CD  F7  95  CD  EC
95  CD  86  96  18  9F  01  09  00  21  E5  96
CD  DF  95  DD  7E  12  E6  0E  CB  3F  FE  00
28  10  FE  01  28  17  FE  02  28  1E  FE  03
28  25  CD  D6  96  C9  01  05  00  21  09  97
CD  DF  95  18  F1  01  06  00  21  0E  97  CD
DF  95  18  E6  01  0C  00  21  14  97  CD  DF
```

```
95  18  DB  01  05  00  21  20  97  CD  DF  95
18  D0  06  06  3E  0D  CD  5A  BB  3E  0A  CD
5A  BB  10  F4  C9  54  79  70  65  20  20  20
20  20  4C  65  6E  67  74  68  20  20  20  53
74  61  72  74  20  20  20  20  42  6C  6F  63
6B  20  20  20  20  42  41  53  49  43  42  49
4E  41  52  59  53  63  72  65  65  6E  20  49
6D  61  67  65  41  53  43  49  49  00  00  00
00  00  00  00  00  00  00  00  00  00  00  00
00  00  00  00  00  00  00  00  00  00  00  00
00  00  00  00  00  00  00  00  00  00  00  00
00  00  00  00  00  00  00  00  00  00  00  00
00  00  00  00  00  00  00  00  00  00  00  00
00  00  00  00  00  00  00  FE  02  C2  2C  95
CD  06  B9  F5  DD  7E  02  CD  A5  BB  DD  5E
00  DD  56  01  D5  DD  E1  06  08  7E  DD  77
00  23  DD  23  10  F7  F1  CD  0C  B9  C9  FE
01  C2  2C  95  DD  7E  00  FE  00  28  11  3E
C3  21  C3  97  32  F1  BD  7D  32  F2  BD  7C
32  F3  BD  C9  3A  29  95  32  F1  BD  3A  2A
95  32  F2  BD  3A  2B  95  32  F3  BD  C9  CD
29  95  D0  CD  5A  BB  C9  FE  02  C2  2C  95
DD  7E  02  CD  35  BC  DD  6E  00  DD  66  01
70  23  71  C9  FE  02  C2  2C  95  DD  6E  00
DD  66  01  22  2F  98  DD  6E  02  DD  66  03
22  31  98  DD  21  70  01  DD  4E  02  DD  46
03  2A  31  98  37  3F  ED  42  28  13  CD  1B
BB  D8  DD  E5  E1  DD  5E  00  DD  56  01  19
E5  DD  E1  18  DE  DD  23  DD  23  DD  23  DD
23  DD  E5  D1  2A  2F  98  73  23  72  C9  00
00  00  00  FE  02  C2  2C  95  DD  6E  02  DD
66  03  7E  32  FB  98  23  4E  23  46  ED  43
F9  98  DD  7E  00  FE  01  28  17  3A  FB  98
5F  AF  57  1B  2A  F9  98  19  7E  C6  80  77
3A  FB  98  FE  01  CC  D9  98  DD  21  70  01
DD  5E  00  DD  56  01  DD  6E  02  DD  66  03
22  31  98  7D  B4  C8  CD  1B  BB  D8  CD  90
98  DD  E5  E1  1B  19  23  E5  DD  E1  18  DC
DD  E5  E1  DD  E5  D5  C1  0B  0B  0B  0B  D5
11  04  00  19  ED  5B  F9  98  C5  1A  BE  28
0B  C1  0B  23  78  B1  20  F0  D1  DD  E1  C9
3A  FB  98  47  1A  BE  20  ED  23  13  10  F8
CD  C5  98  18  E4  CD  EC  95  DD  E5  E5  D5
C5  2A  31  98  CD  F7  95  C1  D1  E1  DD  E1
C9  01  16  00  21  E3  98  CD  DF  95  C9  57
69  6C  6C  20  61  6C  73  6F  20  66  69  6E
64  20  74  6F  6B  65  6E  73  2E  00  00  00
CD  97  99  FE  01  C2  2C  95  DD  6E  00  DD
66  01  E5  22  31  98  CD  06  B9  F5  CD  00
B9  CD  29  99  F1  CD  0C  B9  CD  18  BB  E1
FE  0D  C8  11  08  00  19  18  E1  06  08  11
25  97  2A  31  98  7E  12  23  13  10  FA  2A
31  98  CD  F7  95  3E  0A  CD  6F  BB  06  08
11  25  97  C5  1A  D5  CD  6D  99  D1  CD  7B
BB  E5  7C  C6  1E  CD  6F  BB  1A  D5  CD  5D
BB  D1  E1  7C  C6  04  CD  6F  BB  13  C1  10
DE  CD  EC  95  C9  C5  06  00  4F  CB  1F  CB
1F  CB  1F  CB  1F  E6  0F  FE  0A  30  07  C6
```

30	CD	5A	BB	18	05	C6	37	CD	5A	BB	78
FE	01	28	05	79	06	01	18	E4	C1	C9	F5
3E	02	CD	0E	BC	F1	C9	FE	01	C2	2C	95
CD	97	99	DD	6E	00	DD	66	01	E5	22	31
98	CD	29	99	CD	18	BB	E1	FE	0D	C8	11
08	00	19	18	EC	FE	02	C2	2C	95	DD	6E
00	DD	66	01	22	2F	98	DD	7E	02	CD	C2
BC	D2	0B	9A	ED	5B	2F	98	7E	12	13	23
0B	78	B1	20	F7	C9	FE	02	C2	2C	95	DD
6E	00	DD	66	01	22	2F	98	DD	7E	02	CD
C5	BC	D2	0B	9A	ED	5B	2F	98	7E	12	23
13	0B	78	B1	20	F7	C9	01	17	00	21	15
9A	CD	DF	95	C9	43	61	6E	27	74	20	66
69	6E	64	20	74	68	65	20	45	6E	76	65
6C	6F	70	65	00	00	00	00	00	00	00	00
00	00	00	00	00	00	00	00	00	00	00	00
00	00	00	00	00	00	00	00	00	00	00	00

START ADDRESS 37800 SPLITON/SPLITOFF

21	42	94	01	C4	93	CD	D1	BC	3A	5A	BB
32	3F	94	3A	5B	BB	32	40	94	3A	5C	BB
32	41	94	C9	CC	93	C3	DC	93	C3	E8	93
53	50	4C	49	54	4F	CE	53	50	4C	49	54
4F	46	C6	00	3E	C3	21	F5	93	32	5A	BB
22	5B	BB	C9	3A	3F	94	32	5A	BB	2A	40
94	22	5B	BB	C9	FE	3A	20	1F	F5	3A	3E
94	FE	00	28	04	F1	C3	3F	94	3E	0D	CD
3F	94	3E	0A	CD	3F	94	CD	34	94	F1	3E
3A	C3	3F	94	FE	22	20	15	F5	3A	3E	94
FE	00	28	07	3E	00	32	3E	94	18	05	3E
01	32	3E	94	F1	C3	3F	94	06	05	3E	20
CD	3F	94	10	F9	C9	00	CF	00	94	92	94
C4	93	00	00	00							

START ADDRESS 37600 KEYON/KEYOFF

C3	F7	92	4B	45	59	4F	CE	4B	45	59	4F
46	C6	00	E3	92	C3	01	93	C3	3F	93	21
2C	93	01	EF	92	CD	D1	BC	C9	FE	01	C0
3A	3E	93	FE	01	C8	3E	01	32	3E	93	DD
6E	00	DD	66	01	22	30	93	21	34	93	11
56	93	06	05	0E	FD	CD	EF	BC	21	32	93
CD	DA	BC	C9	00	00	00	00	00	00	00	00
00	00	00	00	00	00	00	00	00	00	00	FE
00	C0	3E	00	32	3E	93	21	32	93	CD	DD
BC	2A	30	93	36	00	23	36	00	C9	CD	09
BB	30	04	CD	6A	93	C9	3E	00	2A	30	93
36	00	23	36	00	C9	2A	30	93	77	23	36
00	3A	06	BB	2A	07	BB	32	94	93	22	95
93	21	97	93	22	07	BB	3E	C3	32	06	BB
11	5E	C4	0E	FD	CD	45	BB	CD	4B	BB	C9
00	00	00	3A	94	93	32	06	BB	2A	95	93
22	07	BB	3E	FC	37	C9	00	00			

Chapter 5
The CRTC and Screen Memory

In this Chapter we'll examine the 6845 Cathode Ray Tube Controller integrated circuit that controls screen output on the Amstrad Computers. We'll also take a look at how screen memory is arranged, and see how it's possible to program these two facets of the system directly. Before we go on, there is no reason why we need to directly interact with these components of the computer. However, it's: a) Educational, and b) Fun!

The 6845

We'll start with this chip. First of all, what does it do? In broad terms, it sets the screen format in terms of character columns and rows, where the screen is to be positioned in memory and is responsible for keeping an eye on the complex timing relationships that are necessary to keep the screen readable! It is helped in this by the Video Gate Array, and these are together controlled by the Firmware Routines. As you will probably appreciate, carelessly fiddling around with the CRTC is a bit like putting sand in a car gearbox; the results are interesting but on the whole undesirable!

The CRTC contains 18 control registers, labelled R0 to R17. These are accessed in the Input/Output map of the system, and the addresses used are as follows:

&BC00 output of Register Number
&BD00 output of Value to Register
&BE00 Read the CRTC Status Register
&BF00 Read a Value from a Register

The BASIC IN and OUT instructions can be used to interact with the CRTC directly, or The |CRTC command that is in the RSX commands we added in Chapter 4. I will use |CRTC when writing to the CRTC.

NEVER read values from &BC00 or &BD00
NEVER write values to &BE00 or &BF00

Of the CRTC registers, R0 to R13 are WRITE ONLY registers, R16 and R17 are read only, and R14 and R15 are read/write registers. The status register contains various bits of information about the CRTC. We'll take a detailed look at all the registers, with programming examples, shortly.

Sending data to or reading data from the CRTC is easy. The register number is set up on &BC00, and the data to be written is set up on &BD00. To use |CRTC, the syntax is simply |CRTC,register, value.

Alternatively, the Register number is set up on &BC00 and data is read with an IN instruction from &BF00. The status register is read directly.

One point to note is that the contents of some of these registers appear to be reset to the values expected by the Firmware when a return to the 'Direct Mode' of operation is made. (When a program finishes executing, for example). Another point to note is that loading certain values to some registers causes the machine to 'lock up', requiring you to turn the computer on and off again! To help you in your programs, the below list gives the usual status of the CRTC registers. There appears to be no difference in register contents in different screen modes.

Register	**Contents**
0	63
1	40
2	46
3	&8E
4	38
5	0
6	25
7	30
8	0
9	7

These values are for UK versions of the Amstrad Computers. The contents of registers 10 to 17 can be altered by the Operating System of the computer during normal operation.

The Screen according to the 6845

The way in which the CRTC 'sees' the screen is quite different from the way in which we do. Before we see how to program the registers, a few words on the terminology of the CRTC. Figure 5.1 shows a diagram of the screen according to the CRTC. You will note immediately that there are two measurements of screen width; the DISPLAYED CHARACTER WIDTH and the TOTAL CHARACTER WIDTH. The 'Character Width' is used as the unit of time in CRTC programming, various registers being programmed in terms of character widths. Vertically, each character is seen by the CRTC as being made up of a number of SCAN LINES on the screen. You can see these if you look closely at the screen.

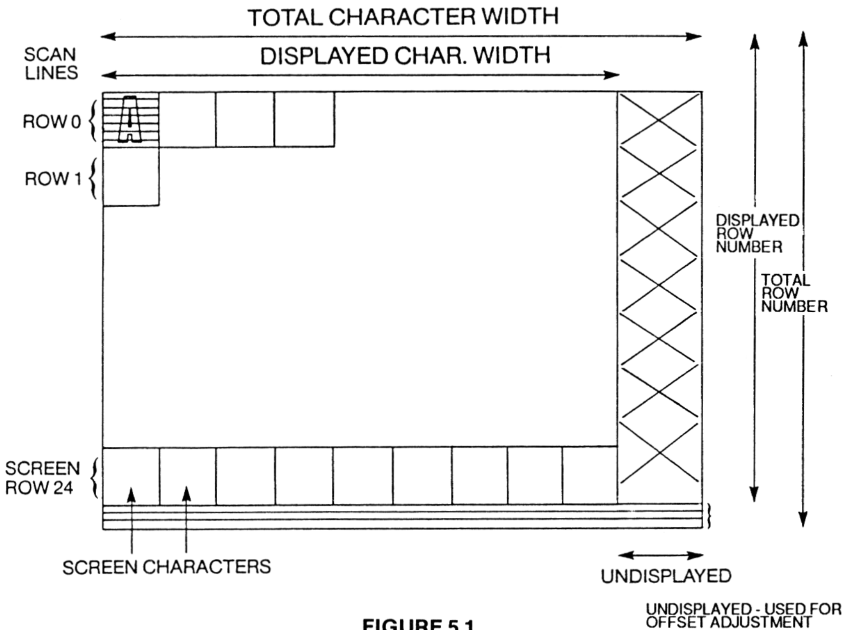


FIGURE 5.1

CRTC Screen Layout.

The 8 lines that make up a character are called a CHARACTER ROW. The Scan Line is used as the basic unit of time for the structure of the screen vertically.

Vertically, we have the DISPLAYED ROW NUMBER and the TOTAL ROW NUMBER. In addition, the CRTC often has a few 'spare' scan lines called the SCAN LINE ADJUST which can be used to move the displayed portion of the screen up or down a character row or two.

The CRTC is also responsible for generating what are called SYNCHRONISATION PULSES. These simply provide signals to the hardware of the display which indicate when a scan line has been completed, when a new one is to start, etc. We needn't go into this in any detail, as the hardware expects these Sync Pulses to be of a certain size and to occur at certain times. If we start messing around with them we will almost certainly cause the screen display to become unreadable.

There are two Sync Pulses; Vertical and Horizontal. They are defined by their WIDTH, and their POSITION. For the Horizontal Sync Pulse the width is in terms of characters, and the position is in terms of character widths from the left hand edge of the screen for a given scan line. For the Vertical Sync Pulse, it s width is measured in terms of 'scan line times', one of which is the amount of time needed for the display to produce a scan line. It s position is in terms of Character Rows.

The only other thing that we need bother about at the moment is the INTERLACE of the screen. The screen is made up of scan lines, split into Odd and Even lines; the screen is refreshed every 50th of a second, but each refresh will affect only the Odd numbered lines of the screen or the Even numbered lines of the screen. It's possible to get the CRTC to write to Even Scan lines (Normal Synchronisation), or both Even and Odd Scan lines. (Interlace Sync. – so called because the two 'fields' of Even and Odd scan lines are interleaved with each other.) Figure 5.2 shows this.

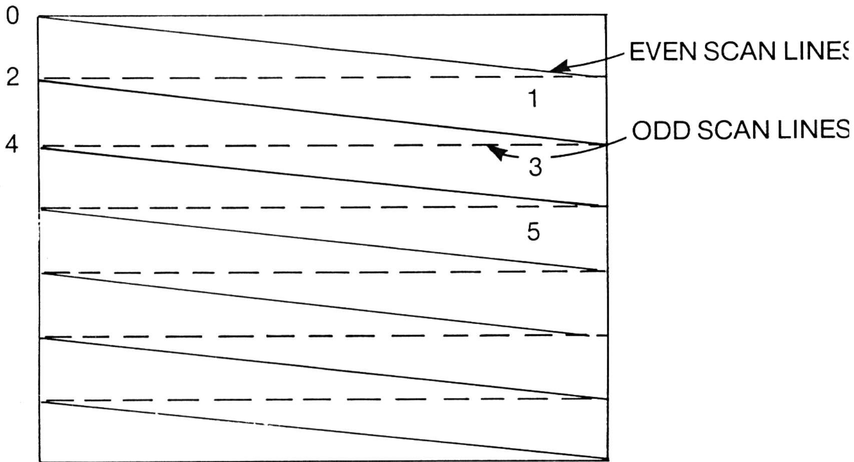


Figure 5.2

Interlaced and Normal Sync.

Finally, Frame Flyback. Once the CRTC has produced a full screen of information, it sends an interrupt to the Z-80 CPU to inform it of the fact. This is rather important, as after the screen of information, or FRAME, has been produced, there is a period in which nothing is written by the CRTC to the screen. This allows the hardware of the display circuitry to get the electron beam which writes to the screen from the bottom right corner of the screen, where scans finish, to the top left of the screen, where scans start. This is called the Flyback Period, and occurs every fiftieth of a second on UK versions of the Amstrad. The Operating System uses the Frame Flyback period to do such things as change inks, or similar operations where it's important not to change things half way through a screen being produced. We can also make use of the Frame Flyback interrupt as a source of timing pulses, as we've already seen, or as a means of getting smoother moving graphics, as we'll see later.

Let's now go on to study the Registers of the CRTC and see what we can do with them.

Before going any further, a few notes about Character Widths. On the Amstrad, a Character Width as seen by the 6845 only corresponds to the character width as seen by our own eyes in Mode 1. In Mode 2, a single character width is used to display two characters on the screen, thus giving an 80 column screen, and in Mode 0 two character widths are required to display a single character, thus giving a 40 column screen.

Register 0

This register controls the Total Number of Horizontal Characters, and is set up to be 63 by the Firmware. R0 is always set to hold one less than the number of Displayed and Non Displayed characters across the screen. This CRTC register isn't terribly useful to us; 63 and 64 are the only values in this register that give a readable screen. Values of less than around 50 cause the computer to lock up, requiring that the system be turned off and on again. Other values cause instability of the display.

Register 1

This register determines the number of Displayed Characters per line. Altering it can produce interesting, but on the whole useless, results.

Register 2

This is a slightly more useful register. It is called the Horizontal Sync. Position Register, and determines the position of the Horizontal Sync Timing pulse in terms of character widths from the left hand side of the screen. Altering the value held in the register alters the position of the start of each screen line on the screen. This has the effect of moving the whole display to the left or to the right. Increasing the value held in this register will move it to the left and decreasing it moves the whole screen to the right. However, there are limits; only values between 0 and 63 are useful. Above 63 will crash the system. The reason for this is that we'd be trying to generate a whole CRTC line of 63 character widths without a Sync Pulse. It doesn't like this. Program 'CRTC2', below, shows this register in use. Note that the CRTC command has been used, so either load the RSX commands into your system or replace the CRTC command with:

```
OUT &BC00,2:OUT &BD00,value%
```

To see the program in use, press the left and right arrow keys. These will move the screen. Try running the program in different screen modes. Note how the program keeps register values to between 0 and 63 to prevent crashes.

```
100 ON BREAK GOSUB 220
110 MODE 0
120 FOR i=0 TO 100
130 DRAW RND*640,RND*400,RND*15
140 NEXT
150 value%=43
```

```
160 a$=INKEY$:IF a$="" THEN GOTO 160
170 char%=ASC(a$)
180 IF char%=242 AND value<63 THEN value%=value%+1
190 IF char%=243 AND value>1 THEN value%=value%-1
200 |CRTC,2,value%
210 GOTO 160
220 |CRTC,2,43
230 MODE 1
240 END
```

Program CRTC2

Once the lines have been drawn, press the arrow keys to see the screen move.

Register 3

This determines the width of the Vertical and Horizontal Sync Pulses, the Vertical pulse being defined by the upper 4 bits of the register and the Horizontal pulse width being defined by the lower 4 bits. Best left alone.

Register 4

This register, aided by Register 5, is used to determine the rate at which the screen is refreshed. It is thus vital for the stability of the display, and very little can be done with it.

Register 5

This register is used with Register 4 to provide some 'fine control' to the refresh rate. Called the Scan Line Adjust register, it controls the number of extra scan lines at the bottom of the screen. It can be used to a small extent to move the screen display 'down' slightly. However, if you do this, any scrolling of the screen will lead to the system locking up. I would suggest, therefore, that you leave this register alone.

Register 6

This is called the Vertical Displayed Character Register, and it determines the number of rows of characters to be displayed on the screen. This is set to 25 on the Amstrad, giving the 25 character rows that we all know and love!

We can modify the contents of this register with safety, producing a screen that has fewer lines displayed. For example,

```
|CRTC,6,20
```

produces a display in which there are only 20 displayed rows of characters. These are the top twenty rows of the display. If you do this, then the CRTC only displays twenty rows but the Firmware still thinks that it's got 25 rows to play with. To get around this, the WINDOW command is used to limit the area of the screen that can be written to and the ORIGIN command is used to set up a graphics window of the same size. The program 'CRTC6' shows this in action.

```
100 REM setting up a 'short' mode 1
110 REM for other modes simply alter
120 REM  the parameters of the WINDOW
130 REM statement. CRTC must be
140 REM implemented, or replace with
150 REM OUT commands.
160 MODE 1
170 |CRTC,6,20
180 ORIGIN 0,80,0,640,400,80
190 FOR i=0 TO 7
200 WINDOW #i,1,40,1,20
210 BORDER 2
220 NEXT
```

Listing of CRTC6

An interesting point to note is that the bottom 5 lines of the display in this modified screen display are given to the screen Border, and their colour is now set by the BORDER command. A practical application for this technique will be described when we look at the structure of the Screen Memory.

Register 8

This is the Interlace and Delay Register of the CRTC and isn't much use to us. Again, altering this register can cause the display to become unstable. If you are interested in having a play, then the register is arranged as follows.

Bits 0 and 1

These control the Interlace Mode to be used by the CRTC. 00 and 10 give normal Sync, 01 gives Interlaced Sync.

Bits 4,5,6,7

These are called Blanking Delay Bits and are used by the CRTC to take account of the amount of time needed by the computer system to transfer data from memory to display circuitry. On the Amstrad, this

delay isn't required, as the system is fairly fast. Thus these bits are all set to zero, as are Bits 0 and 1.

Register 9

This is another register of limited use to us, as it controls the number of scan lines that the CRTC uses to display a character. It is set to n-1, where n is the number of scan lines. The Amstrad screen displays use 8 scan lines to display a character, so the register is set to 7. Altering this value offers us no benefits.

Registers 10 and 11

The CRTC can provide a cursor, and these two registers define the size of the cursor. However, I have been unable to get any useful results from messing around with these registers, and so can offer no suggestions for their use.

Registers 12 and 13

These are very interesting registers, and are used to define the start address of the screen memory. The CRTC treats the contents of these two registers as a 14 bit address which specifies the start of the area of memory to be used as video data. The low byte of the address is in Register 13 and the High 6 bits are in Register 12. We'll see how the CRTC expects to find its Video memory in the second part of this Chapter.

For the time being, however, type in the program 'CRTC12'. RUN it and then use the Cursor keys. This will alter the address used as the start address by the CRTC. The start address refers to the address of the top left byte of the screen. This is usually &C000; in this program we're simply altering the screen address by 40 to move up and down a character row, and 1 to move left or right.

```
100 REM altering CRTC registers 12
110 REM and 13 to move the area of
111 REM memory used as screen memory
120 ON BREAK GOSUB 240
130 MODE 1
140 LOCATE 10,10:PRINT "Hello There"
150 value=&C000
160 a$=INKEY$: IF a$="" THEN GOTO 160
170 char%=ASC(a$)
180 IF char%=240 THEN value=value+40
```

```
190 IF char%=241 THEN value=value-40
200 IF char%=242 THEN value=value+1
210 IF char%=243 THEN value=value-1
220 |CRTC,13,value MOD 256:|CRTC,12,value\256
230 GOTO 160
240 REM break subroutine
250 MODE 1
260 END
```

Listing of CRTC12

When you first press a key, the screen changes and you will see the text that was printed in the lower part of the screen and a pattern in the upper part of the screen. The last line of this pattern is the stack, and you can see how the stack is constantly changing by the way in which the pattern displayed changes. Pressing keys will also cause alteration to the pattern that is displayed. If you press 'ESCAPE' once, you'll see the stack 'freeze' and other parts of the pattern come to life. Try the below as a further demonstration. Again, it's important to remember to have the RSX commands in the machine!

```
array
100 MODE 1
110 ON BREAK GOSUB 210
120 DIM a(2000)
130 PRINT "Press any key to see the"
140 PRINT "Array fill up"
150 CALL &BB18
160 |CRTC,12,0:|CRTC,13,0
170 FOR i=0 TO 2000
180 FOR i=0 TO 100:NEXT
190 a(i)=i
200 NEXT
210 REM Break Key handler
220 MODE 1
230 END
```

Listing of ARRAY

This program allows you to see the array 'a' being initialised. The odd way in which the screen is filled up is due to the arrangement of the Screen Memory, and we'll look at this shortly. The rapidly changing areas, about 1/3rd of the way down the screen are due to the 'i' and 'j' variables.

Registers 14 and 15

These set the position of the CRTC generated cursor, and the contents represent a screen memory. There is little to be gained from altering this register.

Register 16 and 17

These are used when we wish to use a Light Pen with the Amstrad. On the Amstrad computers, there is a connection on the Expansion Bus called L.PEN, and this is connected to the CRTC in such a way that a pulse on it causes the CRTC to load registers 16 and 17 with the screen address that it's currently writing information to. The pulse on the LPEN line is typically caused by a Light Pen which senses the position of the scanning electron beam on the screen by the light generated when the beam hits a spot on the screen. We cannot see this due to the speed at which the screen is scanned, but a suitable sensor will detect this. The address thus strobed into the Light Pen Registers is stored with the high 6 bits in Register 16 and the low 8 bits in Register 17. It's beyond the scope of this book to look at practical Light Pens, however.

Screen Memory

Unlike the CRTC, the way in which the Screen Memory is used varies in different Screen Modes. In addition to the actual 16k of screen memory, there are other areas of RAM used by the Firmware involved in screen handling as workspace. These are described in *'The Amstrad Whole Memory Guide'*, also published by Melbourne House.

The Screen memory of the Amstrad is 16384 bytes long and is mapped into the addresses &C000 to &FFFF. It is overlaid by the ROM, and PEEKs and POKEs to any address in this area of memory will always access Screen RAM. Firmware routines are needed to access the ROM, as can be seen by examining the listing of the 'ROMREAD' RSX command that was described in Chapter 4.

We've already seen how the CRTC uses registers 12 and 13 to tell it where the screen memory is to start. There is a provision in the Firmware for the screen to begin at any of &0000, &4000, &8000 or &C000. However, of these, only the &4000 and &C000 addresses are viable; the others overwrite the Firmware Workspace, Jumpblocks or System Restarts. For the BASIC programmer, only address &C000 is useful, as the other address, &4000, limits the amount of space we've got for a BASIC program to a rather miserly 15k or so.

Of course, if you're writing a large part of the program in machine code, then the area of memory from &C000 can be used.

The screen itself consists of 200 screen lines, each consisting of 80 bytes of memory. Even my poor arithmetic tells me that this comes to 16000, not 16384. There are some areas of RAM in the section put aside for screen memory that are not used. Later we'll see how these bytes can be used for temporary storage. The RAM that is used as screen memory is split into 8 2048 byte blocks, which are addressed as follows.

Block 1	&C000-&C7FF
Block 2	&C800-&CFFF
Block 3	&D000-&D7FF
Block 4	&D800-&DFFF
Block 5	&E000-&E7FF
Block 6	&E800-&EFFF
Block 7	&F000-&F7FF
Block 8	&F800-&FFFF

Each line of the screen is defined by 80 consecutive bytes. For example, the top line of the screen is defined by the first 80 bytes of Block 1. To see this, try the below short program.

```
10 MODE 1
20 FOR address=&C000 TO &C04F
30 POKE address,255
40 NEXT address
50 GOTO 50
```

Line 50 is needed to prevent the 'Ready' prompt from returning. A red line is generated across the top of the monitor screen. Now, you might expect the second line of the screen to begin at address &C050, and run to &C09F. Well, I did, anyway. But it doesn't. Instead, the second screen line is defined by the first 80 bytes of Memory Block 2. This is repeated as shown in Figure 5.3. It is this rather odd arrangement that gave the peculiar way in which the screen filled up in the 'ARRAY' program. The way in which the memory maps on to the display in this odd fashion also makes directly accessing the screen rather difficult. You might like to try the below routine out to confirm the arrangement of screen RAM.

```
10 MODE 1
20 POKE &C000,255
30 POKE &C800,255
40 POKE &D000,255
50 POKE &D800,255
60 POKE &E000,255
70 POKE &E800,255
80 POKE &F000,255
90 POKE &F800,255
95 GOTO 95
```

Again line 95 prevents the prompt from reappearing. This arrangement doesn't alter in different modes. What does, however, is the way in which the 80 bytes that make up the screen line represent characters on the screen. Let's now see how this is done.

Screen Mode 2

We'll start here because this is the easiest screen mode to understand. As with all screen modes, the first byte of the top screen line is only at address &C000 immediately after a Mode change and before any scrolls of the screen take place. Each of the bytes of a screen line represents a single character width, thus bytes &C000, &C800, &D000, &D800, &E000, &E800, &F000 and &F800 will represent the character at screen position 1,1. This can be demonstrated by the below routine. Don't forget to load in the RSX commands from Chapter 4.

```
10 MODE 2
20 |SYMBOL,65,30000
30 add=30000
40 FOR screen=&C000 TO &F800 STEP &800
50 POKE screen,(PEEK(add))
60 add=add+1
70 NEXT screen
80 GOTO 80
```

This simply transfers the bytes that define the letter 'A' to screen memory,line 80 stopping the prompt returning. Each bit of the byte that is set to 1 will be displayed in the INK colour, and a '0' will be in the PAPER colour. There is no other Colour Information involved, as there are only two colours available in Mode 2. Thus we get the 80 column screen, as each bit in the byte can represent a pixel of the character.

Screen Mode 1

This is slightly more involved, as we have 4 possible colours in which a pixel can be displayed. In Mode 2, each byte defined 8 pixels; here, each byte only defines 4 pixels. As each character is 8 pixels wide this reduces the width of the screen to 40 characters. Figure 5.4 shows how the screen characters are defined, and also how bits in the bytes define pixels on the screen. Each character square is thus defined by 16 bytes of video RAM.

Mode 0

This mode has 16 colours, and a byte of memory defines only two pixels, 4 bits being needed to code which of the 16 colours that particular pixel is to be displayed in. This gives us a screen width of only twenty characters, each character being defined by 4 bytes on each screen line. A character in Mode 0 is thus defined by a total of 32 bytes. Figure 5.5 shows how the memory is arranged for Mode 0.

If you were to run the 'TRANSFER' program in Modes 0 or 1 then you would get an image on the screen that was certainly NOT a letter 'A'!

This leads us to one of the big problems in using fast graphics on the Amstrad, and many other popular home computers. That is, the time needed by the Operating System to work out exactly what bytes are to be placed in screen memory for a given character to be displayed in a given colour. In the above case, the bytes read back by |SYMBOL as the definition of 'A' would need to be further processed before they could be put on the screen to represent the character required.

Direct manipulation of the Screen Contents is best done, however, with machine code routines, as these will provide the necessary speed. Examples of such routines can be found in my book '*Ready Made Machine Language Routines for the Amstrad*' published by Melbourne House. However, let's see what we can do without machine code. The first problem is to find out how our usual method of addressing a screen position in terms of row and column can be converted to an actual screen address.

Columns, Rows and Screen Addresses

The address of the top row of a given character position on the screen can be calculated using the below equation.

address=&C000+(pixel*column)+(row*80)

This assumes that there is no OFFSET, which is a slight amendment to the screen start address which can be made by the Firmware and the CRTC. Here 'pixel' depends on the mode; it is 1 for Mode 2, 2 for Mode 1 and 4 for Mode 0. 'column' and 'row' are numbered from 0 upwards. The address returned is that of the first byte of the top line of the character location, and the address of the other lines of the character location can be obtained by adding 2048 to this base address for each screen line. Thus, for example, to put a block of foreground colour at location 20,20 on a Mode 2 screen, we could try the below program. Note that row and column locations are from 0 upwards when we are accessing memory directly.

```
10 MODE 2
20 row=19
30 column=19
40 pixel=1
50 address=&C000+column*pixel+80*row
60 FOR screen=0 TO 7
70 POKE address,255
80 address=address+2048
90 NEXT screen
```

For other modes, the value of 'pixel' would have to be changed. In addition, the bytes that make up the rest of the character width would have to be taken in to account and the colour considerations taken in to account. To see the above working in Mode 1, simply alter the program as follows:

```
10 MODE 1
40 pixel=2
70 POKE address,255:POKE (address+1),255
```

The rest of the program remains the same.

How are bits in a byte mapped on to pixels? Well, it's fairly straight forward, and Figure 5.6 shows how each byte in different screen modes maps on to actual pixels on the screen. Starting with Mode 2, there are only two colours, and so each bit when on sets foreground colour for that pixel and when off sets that pixel to be background colour. For Mode 1:

MSB of Pixel	LSB of Pixel	Colour
0	0	PEN 0
0	1	PEN 1
1	0	PEN 2
1	1	PEN 3

In a similar fashion, Mode 0 maps as:

MSB LSB	Colour
0000	PEN 0
0001	PEN 1
0010	PEN 2
..	...
1111	PEN 15

You can see now why we are limited to a given number of colours in each mode. You might like to experiment with setting different pixels in a character square to different colours, thus giving you multicoloured characters. The program PCOLOUR shows a 'character' that was put together using this technique, and shows how it can be moved around. Obviously, superior results would come from machine code.

```
100 MODE 1
110 row=19
120 add= &C000+row*80
130 FOR i=add to add+79
140 POKE i,RND*255
150 POKE (i+2048),RND*255
160 POKE (i+4096),RND*255
170 CALL &BD19
180 POKE i,0
190 POKE (i+2048),0
200 POKE (i+4096),0
210 CALL &BD19
220 NEXT
```

Listing of PCOLOUR

The CALL &BD19 simply causes processing to pause until a Frame Flyback operation. User of the 6128 may replace these with a FRAME command. Direct access of the memory can thus produce interesting, multicoloured characters. However, for moving graphics that make

use of direct screen access machine code is needed, and that is beyond the scope of this book. We will see more practical methods of animation from BASIC in a later Chapter.

This method of arranging screen memory does make one thing rather difficult, and that is finding the character at a given screen location. In version 1.1 of BASIC, the COPYCHR$ function returns the character at a given screen location. In Version 1.0 the firmware routines will have to be used.

Saving and Loading Screen Memory

The easiest way to save screens of data is as a byte file, with a save command such as:

```
SAVE "screen",B,&C000,&4000
```

assuming that the screen hasn't scrolled since the picture was drawn. Such a file can then be loaded back with a command such as:

```
MODE n:LOAD "screen",&C000
```

The 'MODE' command here serves two functions; that of setting the appropriate mode and that of ensuring that address &C000 corresponds to the top left of the screen.

'Poaching' Screen Memory

It is possible to 'borrow', or poach, some of the Screen Memory and use it for storage of machine code programs, bytes, etc. You can't use memory thus released for BASIC programs, and neither can any machine code placed in such memory call ROM routines that 'live' in the upper ROM.

We mentioned earlier on in this Chapter that some bytes of screen memory are not accessed by the CRTC under most circumstances. These areas of memory are:

&C7D0 - &C7FF
&CFD0 - &CFFF
&D7D0 - &D7FF
&DFD0 - &DFFF
&E7D0 - &E7FF
&EFD0 - &EFFF
&F7D0 - &F7FF
&FFD0 - &FFFF

These areas are, however, used by the Video Circuitry when a scroll of the screen takes place. For example:

```
10 MODE 1
20 FOR screen=&D7D0 TO &D7FF
30 POKE screen,j
40 j=j+1
50 NEXT screen
60 CALL &BB18
70 FOR screen=&D7D0 TO &D7FF
80 PRINT PEEK(screen)
90 NEXT
```

This will poke a series of numbers into Video RAM; observation of the screen reveals no change. Line 60 simply waits for a key to be pressed before the same addresses are peeked out. You will see that the values are unchanged -1,2,3...etc.

However, if we alter the program by putting in:

```
65 FOR scroll=0 TO 30:PRINT:NEXT scroll
```

will result in the screen scrolling and the bytes POKEd in lines 20-50 of the above program are lost as the screen memory is used by the scroll. This is obviously an unsatisfactory method of using the 'spare' video memory. What we need is some means of stopping the CRTC actually accessing the memory. We can do this by using the CRTC registers, as we'll now see.

We'll use CRTC register 6, the Vertical Displayed Register, and modify it to alter the number of character rows displayed. We then set Graphics and Text Windows up to limit the screen to the appropriate size. The program listing below does this. Note that the Mode change is important to initialise the screen memory to start with &C000 at the top left of the screen. Before going on, the program listing. Again, I've assumed that you've loaded the RSX commands from Chapter 4.

```
10 |CRTC,6,20
20 MODE 1
30 scrwidth=40
40 FOR w=0 TO 7
50 WINDOW #w,1,scrwidth,1,20
60 NEXT w
70 ORIGIN 0,80,0,640,400,80
80 CLS:CLG
```

The 'scrwidth' variable in line 30 should be set according to the mode. (20 for Mode 0, 40 for Mode 1 and 80 for Mode 2). Lines 40 to 60 set up the seven text windows to a new default size of only twenty lines. Line 70 sets the graphics window and origin in a similar fashion.

If you run this, a BORDER 2 command will clearly show the new screen size. Now to the memory that we've freed by this process. The first thing to note is that it is NOT a continuous block of memory; this is due to the way in which the screen memory is arranged. The memory available is arranged in a series of 80 byte 'chunks', some of which do follow each other in the memory map. To see what addresses these chunks have, run the below program. The addresses printed are the addresses of the first and last byte of each block, respectively. Add the lines to the above program.

```
100 FOR row=20 TO 24
110 addr=&C000+row*80
120 j=0:FOR i=0 TO 7
130 PRINT HEX$(addr+j,4), HEX$ (addr+j+79,4)
140 j=j+2048
150 NEXT
```

These blocks of memory can be used to store short machine code routines, data, etc. The addresses given above are only correct if the Mode was set in the above way, but are correct for any screen mode. If you do use this technique, take care to ensure that you don't write or draw outside the area set by the Windows. The results could be rather messy, especially if you're storing machine code in the freed space!

You will notice one peculiarity about the screen when used in this way; any scrolls take place as normal but seem a little slower than usual. Try listing the above program a couple of times to see what I mean. This should cause no problems, however.

Chapter 6
AFTER and EVERY

Locomotive BASIC is one of the few dialects of BASIC that offers us interrupt facilities in BASIC; in most home computer systems, the programmer who wishes to use interrupts is forced into machine code, and even then has difficulties in implementing the facilities required and interfacing the machine code to BASIC. In this Chapter I will examine the AFTER, EVERY and associated commands, discuss applications and also have a brief look at what makes them 'tick'. (If you'll pardon the pun . . .)

AFTER

The Syntax for the AFTER command is:

```
AFTER time GOSUB m
AFTER time,timer GOSUB m
```

where 'm' is the line number of the start of a subroutine to which control is to be passed 'time' fiftieths of a second after either of these commands have been issued. AFTER is a 'one off' interrupt; it will cause the subroutine in question to be entered once only. Of course, a subroutine can issue another AFTER command:

```
 10  AFTER 50 GOSUB 100
 20  PRINT I:I=I+1:GOTO 20
 30  END
 40  :
100  SOUND 1,100
110  AFTER 50 GOSUB 100
120  RETURN
```

but this is the sort of thing that the EVERY command is for! This program will 'beep' every second while the program is running.

The 'timer' parameter is an optional integer between 0 and 3. There are 4 timers available to AFTER, and each of them could be set to activate a different subroutine after different lengths of time. The default timer, and the one we used in the above example, is 0. Timer '0' is the lowest priority timer, and Timer '3' the highest priority timer. If it should pass that two AFTER commands should time out at the same instant in time, then the subroutine attached to the highest priority timer will be executed first, followed by the other routine.

There are a couple of other commands and functions related to the AFTER command.

DI and EI

These Disable and Enable the 4 timers respectively. DI will ensure that no AFTER events are recognised by the system until one of three occurrences:

1. A RETURN command is executed.
2. An EI command is executed.
3. RUN is executed.

The prime use for DI is to ensure that 'time critical' portions of BASIC programs are not interrupted by AFTER or EVERY commands. You might, therefore, execute such a command as the first operation in a subroutine that has been entered by an AFTER or EVERY command. This will ensure that your interrupt subroutine isn't interrupted! Remember that the act of entering a subroutine under control of one of the Interrupt Timers will disable any timers of the same or lower priority for the duration of the subroutine. Thus important AFTER and EVERY commands should use the highest priority timer.

EI simply reverses the effect of DI.

REMAIN(n)

This function returns the number of fiftieths of a second left on a particular Timer, 'n'. It also stops that particular Timer.

The Effect of Break on AFTER and EVERY

Pressing ESC generates an interrupt that is of a higher priority than any of the Timers. It cannot be disabled by DI, although we've already seen an RSX command in Chapter 4 that will disable this event.

However, when ESC is pressed once, the program pauses but the AFTER and EVERY events keep being 'kicked' by the Operating

System. Pressing any other key than ESC at this point will cause program execution to continue, starting with the execution of all outstanding AFTER and EVERY subroutines. This means that you can have several seconds, or longer, in which the Amstrad is simply 'making up' for the time lost between the two key presses.

If we have a full Break Event, then if we are able to restart the program with CONT, any 'pending' AFTER and EVERY events will be processed in the same manner.

EVERY

The EVERY command is a repetitive AFTER command, ensuring that a given BASIC subroutine is entered EVERY now and again. EVERY again counts in fiftieths of a second; in fact, it uses the same timers as the AFTER command, and what we have said about REMAIN, DI, EI and priority applies to EVERY as well as AFTER. The fact that the same timers are used can cause problems to the unwary; if we set up, say, an AFTER and then set up an EVERY command on the SAME timer, the AFTER will be forgotten. Only the EVERY interrupts will be obeyed. The same applies, of course, if we set up an EVERY before an AFTER on the same channel.

The syntax for EVERY is:

```
EVERY time GOSUB m
EVERY time,timer GOSUB m
```

We will see a couple of applications of the EVERY command shortly.

General Applications and Problems

Of the two, EVERY finds the most use in my programs. We can use it to cause a given sequence of operations to be executed every so often. For example, in a games program, we could enter a subroutine every tenth of a second to read the keyboard. However, before we get too carried away with schemes for interrupt driven programs, remember that every second spent in an AFTER or EVERY subroutine is a second spent away from the main body of the program. In extreme cases, with very frequent interrupts, this can lead to a serious slowing down of the program.

On a similar theme, don't make the interrupt subroutines too long. To have a routine that takes 2 fiftieths of a second to execute called every fiftieth of a second by an EVERY command is a little silly; at best it will be executed once every 1/25th second, and would probably slow up the main program to a standstill!

In this case, you would need to either increase the time between successive entries to the subroutine by altering the EVERY command or shorten the subroutine's execution time.

A further point to note is that INPUT appears to cause the timer action to be suspended, much like when ESCAPE is pressed once.

Some Applications

Let's now see how we can use these commands. The first application simply uses AFTER as a timer, without bothering about its interrupt facility. We are using it here to time subroutines:

```
  10   AFTER 32767 GOSUB 1000
  20   FOR I=0 TO 100
  30   PRINT I
  40   NEXT I
  50   GOSUB 1000
  60   END
1000   PRINT 32767-REMAIN(0)
```

Here we are timing the routine in lines 20-40. Line 10 starts the timer, and line 50 stops it. Line 1000 then stops the timer by using REMAIN and prints the time that has lapsed in fiftieths of a second.

AFTER could be used in a similar way in other programs. 32767 is used to give a nice long delay before the interrupt would fire off. The principle could be used for reaction timing (possibly in conjunction with the ON KEY command of Chapter 4), or other timing applications. A second use might be to add a 'password' to a program. We could save a program as a protected program, which contains the following lines:

```
    1 ON BREAK GOSUB 10000
    2  AFTER 200 GOSUB 10000
    3  a$=INKEY$:IF a$="" THEN GOTO 3
    4 IF ASC(a$)<>65 THEN GOTO 3
    5 a=REMAIN(0)
    6 :
    7 REM start of program
10000 NEW
```

This routine is quite simple. Line 2 sets up a 4 second delay before line 10000 is executed, zapping the program. As it's not common to accidentally press ESCAPE twice, line 1 does the same on Break! Line 3 waits for a keypress, and line 4 checks it to see if it's the correct

password, in this case "A". If it is, then line 5 turns off the AFTER. Otherwise, control goes back to line 3. Further security could be introduced by repeating lines 3 and 4 as often as you want, only executing the REMAIN command when a two or three letter password has been entered. INPUT isn't used because it would:

1. Display any letters entered on the screen.
2. Allow the user to break in at the INPUT stage.
3. Disable the timer for the AFTER command while input was awaited.

EVERY can be used to provide a 'Dynamic' view of memory locations, PEEKing out and printing the contents of memory locations repeatedly. The below program does this:

```
  10  EVERY 50 GOSUB 1000
  15  MODE 2
  20  a%=a%+1:GOTO 20
1000  REM EVERY routine
1010  LOCATE 1,1
1020  FOR I%=0 TO 10
1030  PRINT I%+@a%,PEEK(I%+@a%);"  "
1040  NEXT I%
1050  SOUND 1,200
1060  RETURN
```

As it stands, the display will be updated every second; you can reduce the 'time' parameter in line 10 down to 26, thus updating the display every half second or so. The program will print out the 11 bytes starting at the memory locations used to store the variable 'a%'.

The main function of this program, though, is to show one of the problems that the EVERY user can encounter. If we change line 10 to:

```
10 EVERY 20 GOSUB 1000
```

then the EVERY routine is re-entered repeatedly, as indicated by the 'beeping'. However, the routine is entered so often that the main program doesn't have time to do anything else, as indicated by the fact that the bytes of 'a%' do not change.

An obvious function for EVERY is some sort of 'clock', which displays the time while a program is running. I've always thought this a rather redundant application, given the cheapness of digital watches! However, here are the 'bare bones' of a subroutine to implement such a function.

```
   5  MODE 1
  10  hours=10:minutes=30:timer$="12:00"
  20  CLS
  30  WINDOW #7,32,40,1,1:PAPER #7,3:PEN #7,1: CLS#7
  40  EVERY 3000 GOSUB 1000
  50  REM rest of program
  60  :
  70  :
1000  REM Clock subroutine
1010  minutes=minutes+1
1020  IF minutes=60 THEN minutes=0: hours=hours+1
1030  minute$=RIGHT$(STR$ (minutes), LEN (STR$
      (minutes))-1)
1040  IF hours=24 THEN hours=0
1050  hour$=RIGHT$(STR$ (hours), LEN (STR$ (hours))-1)
1060  IF LEN(hour$)=1 THEN hour$= "0"+hour$
1070  IF LEN(minute$)=1 THEN minute$= "0"+minute$
1080  PRINT #7:PRINT #7, hour$;":";minute$;
1090  IF timer$=(hour$+":"+minute$) THEN FOR I=0 TO 200
      STEP 10: SOUND 1,I: NEXT I
1100  RETURN
```

The program is fairly simple: window 7 is used to print the time, and line 10 sets up the start time on the 24 hour clock. The string 'timer$' allows a simple 'alarm clock' function to be implemented. It should be set to hold the 24 hour time (e.g. 'timer$="07:05"' for five past seven) and then when this time is reached a sound will be generated. The subroutine is kicked every minute by the EVERY command. If you want, you could kick it every second, by modifying the EVERY command and adding lines to increment a second counter in a similar way to that in which we've incremented the minute counter. Lines 1030, 1050, 1060 and 1070 serve to convert the 'hours' and 'minutes' variables into strings of the format required.

How AFTER and EVERY work

It is clear that AFTER and EVERY are somehow controlled by the Events System of the Amstrad. In this section, we'll do some detective work that will lead to a full description of how AFTER and EVERY interact with the Operating System. Note that the 'gory details' will be specific to the Amstrad CPC464; any addresses given will be for that machine.

The fact that AFTER and EVERY on UK machines measure their time delays in fiftieths of a second point to either the Frame Flyback or

the Ticker Events being involved. The fact that the timers used by the commands still use fiftieths of a second in other machines points to the Ticker Event as the driver for AFTER and EVERY.

On the 464, the start of the 'Ticker List', which contains all the Event Blocks to be kicked by the Ticker Event, occupies locations &B190 and &B191. Normally, with no Ticker Events running, this address is zero. However, after the execution of a line such as:

```
10 AFTER 32767 GOSUB 1000
```

the address held in these locations is &AC5C. This confirms that the Ticker Event is responsible for controlling AFTER and EVERY. It therefore follows that each AFTER or EVERY command sets up a Ticker Event Block.

Ticker Event Block

A typical Ticker Block is shown in Figure 5.1.

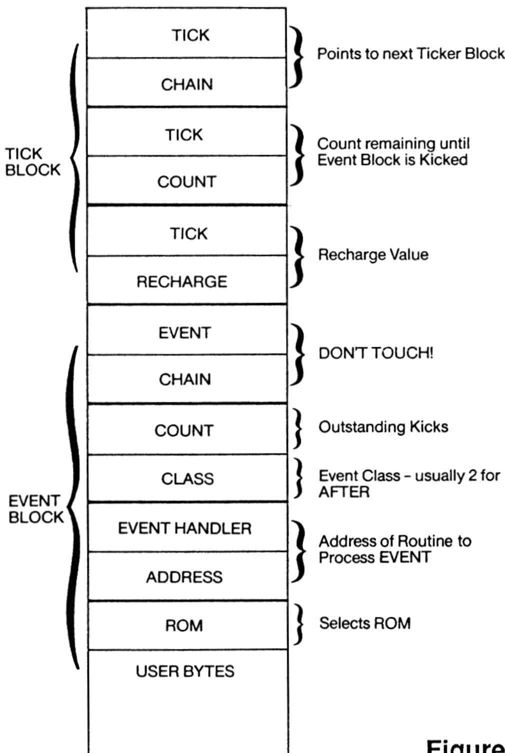


Figure 5.1

So, we now set up an AFTER command as a program line, and investigate the contents of &AC5C and onwards. I used the RSX |DUMP command to do this.

```
  10 AFTER 32767 GOSUB 1000
  20 |DUMP,&AC5C
  30 GOTO 30
1000 RETURN
```

This program was typed in after a hard reset of the machine. The following were revealed:

Ticker Count:	Depends on AFTER command.
Event Class:	&02
ROM select:	&FD
Address of Event Handler:	&C879

In addition, there are 5 extra bytes after the end of the Event Block that appear to be concerned with the AFTER command. I assumed that somewhere in these bytes would be either the line number referred to by the AFTER statement. In this assumption I was partially right. The line number isn't stored as such, but an address referring to that line. Of the 5 bytes, I haven't determined the significance of the first three bytes but the last two hold what appears to be the address of the line number which is the destination of the GOSUB command in the AFTER statement, minus 1. Thus for:

```
AFTER 300 GOSUB 1000
```

if line 1000 started at address &0135 then these two bytes would be found to hold &0134. To see if this supposition was true, I tried the below program. Type it in EXACTLY as it is written, as the two bytes POKEd in in line 2 refer to an address within the program.

```
   1 AFTER 200 GOSUB 1000
   2 POKE 44140,&BE:POKE 44141,&1
   3 GOTO 3
1000 PRINT "Line 1000"
1010 END
2000 PRINT "line 2000"
2010 END
```

RUN the program, and you will see after 4 seconds the message 'Line 2000' printed to the screen rather than the expected 'Line 1000'!. Thus our supposition about these bytes was correct. If you look back at the BASIC line structure shown in Chapter 3, then you will see that this address is the address of the end of line marker ('0') for the line immediately before the one referred to by the AFTER statement.

If you want to experiment with poking different values into the locations mentioned above, there are a couple of points to note.

1. These addresses are for the first AFTER or EVERY declared in the program.

2. The addresses given above are for the Amstrad 464 only.

3. You will need to find the address of the BASIC line which you want to POKE in. To do this, you can use the RSX FIND command, and subtract 5 from the value returned.

Although the above details were ascertained for the AFTER command, they are very similar for EVERY. The only real difference is in the Ticker Block Recharge Count. This two byte parameter is accessed by the OS when the Ticker Count reaches zero. The 'Count' bytes are 'recharged' from the 'recharge' bytes, thus starting the whole thing off again. For EVERY, the recharge bytes are non-zero, the actual value depending upon the EVERY command which set up the block. For AFTER, which is a one off timing event, the recharge bytes are zero.

This leads us rather nicely onto the fact that once a Tick Block has a zero Tick Count, it is not serviced by the OS but is still taking up space and the computer has to spend a little time checking whether the block does need servicing. I am not sure whether an AFTER block is deleted from the Ticker List when it's been kicked.

One final point. Although POKEing the AFTER or EVERY block is possible, as we've seen above, it can be a little hairy! Think what could happen if we POKEd one value into, say, 44140, but before we could POKE the next value the Event Block was kicked. The address held in the Event Block would not be that of the desired line number, and would probably lead to a messy crash!

It's also possible to PEEK out values from these blocks; if we have just one AFTER running, we can have lines like:

```
  10  AFTER 32767 GOSUB 1000
  20  PRINT PEEK(&AC5E)+256*PEEK(&AC5F)
  30  GOTO 20
1000  RETURN
```

This will print out the count remaining on the AFTER timer, just like REMAIN, in fact. However, unlike REMAIN it doesn't stop the clock.

So, what use can we make of this information. Well, just as we 'hijacked' the Break Event in Chapter 4, we could use the AFTER/EVERY Event Block to allow our machine code routines to pass control to a BASIC line by simply setting up an Event Block containing the details above, with the address of our line number, and then kicking that Event. Alternatively, we could hitch such an Event Block up to our 'ON KEY GOSUB' routine, so we don't use the Break Event. Or, what about adding a fifth AFTER/EVERY timer, accessed by RSX commands. You might think that we could add a 'fast' AFTER/EVERY command that uses the Fast Ticker. This wouldn't, though, be that efficient, because even with the normal ticker we still have problems finishing our subroutines before the next event is kicked if we have an EVERY that is trying to call the subroutine very frequently. A Ticker six times as fast would allow a little more 'resolution' in the timing, but even the accuracy would be wasted to some degree as BASIC isn't really fast enough to make the best use of it.

A further application might be to add AFTER and EVERY commands that mimic the already existing ones but provide you with an additional timer. In addition, a new |REMAIN function could be added which could return the amount of time remaining by getting the contents of the 'Tick Count' bytes, without stopping the clock. It should also be possible to alter 'recharge' values while a count down is in progress, and so on.

However, all these are a little beyond the scope of the book; after all, we're supposed to be discussing advanced BASIC, not advanced machine code! However, I hope that this information has given you a few ideas as to further ways in which machine code and BASIC can interact.

Chapter 7
Some Hints on Programming

In this Chapter I want to look at some useful subroutines that haven't really fitted in to any other Chapter. In addition, I'll also look at general programming style.

Program Design and Programming Style

Whenever I write a program I start with a pile of paper, a pencil, and copious amounts of coffee. I don't go to the computer until I've done quite a lot of work. The first requirement is that you have a clear idea in your own mind about what you want the program to do. It's not a good idea to allow a program to 'grow like topsy' as you're actually writing it. Any facilities that are to be in the final program should be designed in to it when you're actually planning the program; failure to do this can lead to problems when the program is running.

Once you have such a 'specification', you can begin to design the program, still without touching the computer. I use a technique known as 'Top Down' design. The specification is split into a series of tasks that the program must perform, and each task is then split into further sub-tasks. This process carries on until the description of each task is detailed enough for you to write the program in whatever language you choose. The description of each task is often done in a 'pseudo language', which describes the operations to be carried out. For example:

```
PRINT"Please press a key"
WHILE key-is-not-pressed
DO
END
```

This simply tells us that the program would wait until a key is pressed. In this particular case, the 'pseudo language' description is very close to a Locomotive BASIC routine. In other cases, there might be a big difference between the p-language description and the BASIC routine. Once we've got a 'p-language' description for each part of the program, we can begin to write the routines.

We now go backwards, in effect. We write the small sub tasks as BASIC functions or subroutines, starting with routines that might be called by many parts of the program. An example might be a routine that waits for a key to be pressed on the keyboard. These routines, I call Primitives. Each such primitive can then be tested independantly from the rest of the program, and you can thus debug your program as you are writing it by debugging the small subroutines. This means that when you write the larger sections of a program, you can at least rely on the small sections you've already tested.

This process is made a little difficult on the Amstrad due to the absence of what are called LOCAL variables. These are variables that are only operative within a subroutine, not in the main body of the program. For example:

```
1000 REM will not work on Amstrad
1010 J%=0
1020 GOSUB 2000
1030 PRINT J%
1040 END
2000 LOCAL J%
2010 J%=2
2020 PRINT J%
2030 RETURN
```

On a system supporting Local variables, the PRINT statement in line 2020 will print out '2' and that in line 1030 will print out '0'. It is as if two separate variables of the same name are maintained by the system. The problem that lack of this ability on the Amstrad causes is that we might write a subroutine that affects the value of a variable that is used elsewhere in the program! The subroutine would thus cause a bug that might be many lines of code and several seconds or minutes away from the subroutine call that set the problem up – a 'time bomb' in the program, if you like. The only way around this is to use care when naming variables for use in subroutines, only use them in subroutines and don't use them in the main body of the program. You in Chapter 6, for example, xp% and yp% turned up everywhere; these are two of my 'pet' names for subroutine variables.

Of course, you will have to set up the variables used in the subroutine before calling the subroutine.

Again, the programs in Chapter 6 show many examples of this. The practical result of following this method of program design is that a program may consist of a series of GOSUB calls in it s first few lines, each subroutine call doing a particular task in the program. For example, the first may initialise arrays and set up variables to initial values, the second print up the rules of a game and prepare the game screen, and a third may allow the user to play the game. This allows you to easily track down bugs that occur; just isolate them to the subroutine that does the job, then find the particular routine that causes the problem. More on debugging later.

Style and the Subroutine!

The way in which programs are written is obviously dependant on the programmer, but there are various rules which can help him develop a good 'style'. Just as in English, BASIC has a 'grammar', which tells us how commands can be used. In addition, however, there are stylistic points which might not help the program run any faster but make the logic of it much easier to follow, thus allowing easy alteration and debugging. Even if you only write programs for yourself, it's a good habit to get into. Two or three months away from a program can lead to the type of mystification usually attributed to 'experimental' films when you come to read the program again. I know – I've experienced both!

Seriously, though, a well written program can remove the need for explanatory notes, and makes life easier.

REMs. We saw in Chapter 1 that REMs take up space. It's a nasty fact of life that well structured and documented programs are very heavy on memory. For this reason I often keep TWO versions of my programs; one containing all the REMs and with well spaced subroutines, and another run-time version which crams the program into as little space as possible. I consider the following to be the principle uses of REM statements.

1. At the top of the program, giving date, version and name of program, together with your name and address, and copyright notice. In addition, put in any system requirements – i.e. BASIC 1.1, disc drive, printer etc. This is especially important if you intend selling the program. Documentation can easily get separated from programs in the commercial environment, so make sure that at least some of your details are in the program.

2. Any 'odd' bits of code should be REMmed for later use. The rule here is 'If it ain't clear what it does- REM it.' This can be useful to you at a later date when you want to use routines from the program in another program.

3. Subroutine 'headers'. These give details about the requirements of the subroutine. The REMs here will detail what the routine does, what variables are used for what purposes, and any special points about the subroutine.

4. File Structure. If the program uses files, then I find it useful to include a REM statement in the File handling subroutine that contains information about how the program expects data to be stored in files.

Variable Names

Again, memory considerations and clarity conflict. A variable name should be descriptive of what the variable represents. For example, in a market garden data base (!) we could have the line:

```
1000 carrots=2000
```

which is clearer than:

```
1000 c=2000
```

The first obviously refers to a property of carrots – probably the number that are available. The second could refer to anything.

In addition, I use certain variables for particular jobs. I%, J%, K% tend to be used as loop control variables in FOR...NEXT loops. xp%,yp% are used as coordinates, and so on.

Thus I can often see at a glance exactly what a part of a program refers to. A similar thing can be done with array names.

Whilst on the subject of variables, I like to try and define them all explicitly. Locomotive BASIC encourages lazy thinking in one of its aspects, in that a variable that is not assigned a value BUT is used assumes the value of '0' or an empty string rather than signalling an error. This can lead to some funny errors on occasion, so always set variables to start values in your initialisation subroutine. In addition, I find it a good idea to explicitly DIM the arrays to be used, rather than rely on the fact that Locomotive BASIC doesn't require you to DIM arrays with less than 10 elements.

FOR...NEXT loops

Where loops are nested to any great depth, I like to follow each NEXT with the variable to which it refers. This makes the program easier to follow. FOR...NEXT loops are used to carry out a sequence of instructions a known number of times. However, it is possible to exit a FOR...NEXT loop before the full number of passes is achieved, and exit the loop in a reasonably 'clean' way. For example:

```
990  found=0
1000 FOR I%=1 TO 100
1010 IF file$(I%)="Fred" THEN found=I%
1020 NEXT I%
```

This routine is simple enough; check each entry in the array 'file$' and then set the variable 'found' to hold the element which is equal to 'Fred'. Now, as it stands, more than one occurence of 'Fred' will result in 'found' holding the element containing the last occurrence of 'Fred'. If you wanted the first occurrence, then there are two ways of doing it. Replace line 1010 with:

```
1010 IF found=0 AND file$(I%)="Fred" THEN found=I%
```

or, exit the loop. The latter is often the best solution, as the above method still requires that all 100 entries are searched. To exit the loop, replace line 1010 with:

```
IF file$(I%)="Fred" THEN found=I%: I%=101
```

This will cause the loop to terminate the next time NEXT I% is encountered, as I% will be greater than the loop termination value in the FOR statement. GOTO to get out isn't a good idea. Neither is the below structure:

```
1000  FOR I%=0 TO 200
...

1100 IF I%>100 THEN GOTO 20
        ...

1500 NEXT I%
1510 RETURN
        ...

2000 NEXT I%
2010 RETURN
```

This might be alright on small loops, but can lead to confusion on longer loops.

And so on to a subject that causes much wailing and gnashing of teeth in computing circles. The GOTO statement. BASIC needs it, but the general rule is to use it as little as possible, and then over small ranges of lines. Don't have GOTO statements sending control all over the program. This is known as 'Spaghetti Programming', because the resultant program is more convoluted than a bowl of spaghetti!

This is reduced by using the Top Down approach of design, as each subroutine can be written as a self contained unit. As most of these routines are short, GOTO's can in my opinion, be used with safety within the confines of the subroutine. By the way, a subroutine longer than a couple of 50 or so lines of code should be given a quick examination; see if there aren't smaller tasks that the subroutine could be split into. On the Amstrad, WHILE...WEND is available, and this can be used in some places instead of GOTO. However, I find GOTO provides greater clarity IF USED WITH care in many situations. The emphasis, here, of course, is on 'if used with care'. So, some simple GOTO rules.

1. Don't have masses of GOTOs going all over the program.

2. Keep the line containing the GOTO and the destination line within a few lines of each other, certainly within a screen full of code. Only break this rule in dire emergency. (World War Three, etc.)

3. Before using GOTO try and think of other ways of achieving the same end.

GOSUB...RETURN

A subroutine can be entered at any line within it, not just the first line of the subroutine. This isn't good practice, though, as it makes the program less clear to read. It usually indicates that a subroutine has two functions. Split it into two and just let each subroutine have one entry point. Similarly, a subroutine can have any number of RETURN statements to terminate it. I like to try and use just one RETURN, even if it is jumped to in some cases by a GOTO statement.

ON...GOTO, ON...GOSUB.

These structures can get rid of the many IF...THEN statements that extensive use of menus can produce. Again, they clarify the reading of a program.

General Program Structure

I try and make my larger programs conform to a general structure in which the first few lines are REMs, followed by a few subroutine calls that make up the body of the program. The definitions of these programs I usually start at around line 1000, and I try and keep all the 'primitive' subroutines in lines above 20000. DATA statements I try and put near to the subroutines that use them, or put at the end of the program.

Error Traps and Error Prevention

This section is of most use to those of you writing programs for that difficult to please animal, the 'computer illiterate user', who will no doubt find every single bug in your program, and will take fright at such messages as 'Subscript out of range' appear as the program grinds to an embarrassing halt. You will no doubt get rid of such errors as 'Syntax Error' or other such programming errors, but don't forget that the user may type in some data that gives the program a severe headache. In this section, we'll see how this can be prevented, starting with deliberate attempts to get in to the program.

BREAK. If you don't want a program to be broken into, use ON BREAK CONT, ON BREAK GOSUB, etc. to prevent this occurrence.

Protected Files. A program saved as a 'P' file can only be run from tape, not loaded and listed.

Input to the Program

Programs usually accept an input from files or keyboard, and here is where problems can arise. Beginning with the keyboard, no keypress should be accepted that the program cannot deal with. The practical upshot of this is that for programs designed to be used by non experts the use of INPUT is rather amateur. The Input Routines featured later in this Chapter show how to do it properly. When waiting for a key to be pressed, use a routine such as:

```
1000 a$=INKEY$
1010 IF a$="" THEN GOTO 1000
```

If you are waiting for a specific key to be pressed, then a little more sophistication is needed. The INSTR function makes life easier here. For example, a common requirement is to wait until Yes/No response is obtained by waiting for 'Y' or 'N' to be pressed. The below routine does this.

```
1000 a$=INKEY$
1010 IF a$="" THEN GOTO 1000
1020 response%=INSTR("YyNn",a$)
1030 IF response%=0 THEN GOTO 1000
1040 RETURN
```

Lines 1000-1010 wait for a key to be pressed, then 'response%' is set up with a call to the INSTR function. If this is 0 then the process is repeated. Otherwise, response% will have the value 1 or 2 for a Yes response or 3 or 4 for a No response. Note how we check for both upper and lower case letters here.

For numeric and string inputs, it's best to use an input routine like the ones listed later in the chapter. These tend to return strings, which can be used as they are or further processed. For example, a string representing a number can be converted into a number using VAL. This might be a record number in a data base, so the first thing to do is to make sure that the number returned is a legal record number. If there are only 100 records in the data base, and a value of 101 is entered, then this would cause an error. Therefore, we refuse to accept the number and ask for another one. The user can be informed of why the number is unacceptable, if desired. This does, of course, mean more work for the programmer but we end up with a more 'robust' and 'user friendly' program. Such 'data validation' can also make it impossible for a user to enter a string when a number is expected, and so on.

If this 'data validation' is effective, then your program will not crash due to incorrect user inputs.

Other validation might include ensuring that data entered for use as a file name is not too long, that data entered into a database program isn't too long for the space available, and so on. Reading data from a file that is not the correct type for the program will lead to a variety of errors, including 'Type Mismatch'. The best way out of this problem is to use the ON ERROR trap to enter a subroutine to indicate that the file type is wrong to the user.

ON ERROR GOTO

With all your programming skill, there are some conditions that are almost impossible to get around using data validation. So, we trap these with an ON ERROR GOTO command which directs control to a given line number when an error occurs. If possible, the error 'trap' routine, as it's called, should clear the error and use RESUME to get things going again. The user should be informed of the error condition. (The system variable ERR holds the error number. On the

whole, line numbers at which the error occurred are useless unless the program is still being tested and it's expected that some 'Syntax Errors', etc. might still show. Users might get a little worried at a plethora of line numbers!) If the error condition is such that you cannot RESUME, then finish the error trap routine with a GOTO statement that passes the user back to a menu, etc. after the error has been cleared.

Debugging Routines

A nicely written routine is also easier to debug than routine written using meandering GOTO statements and meaningless variable names. In this section, I want to look at some of the more common problems that you may encounter.

Syntax Errors

These are usually caused by incorrect use or spelling of BASIC commands. It is a good idea to enter all commands in lower case; if correct the BASIC interpreter will list the statements thus entered in upper case, thus making it easy to see any BASIC keywords that have been entered incorrectly. This also allows you to see those occasions when you've entered a keyword as part of a BASIC variable name.

Additionally, this error can be caused by the number of opening and closing brackets being different. For example:

```
PRINT (((a+b)+2)
```

will give a 'Syntax Error' due to there being three opening brackets and two closing brackets.

The error is also caused by typing in absolute rubbish!

Memory Full

This can be caused by a variety of things. The most common cause is while you are manipulating the value of HIMEM, possible to reserve space for machine code routines. If you try and set the value of HIMEM too high or low then this error can be generated. The opening of a file or use of SYMBOL AFTER will affect the values of HIMEM that are 'legal', and so this should be taken into account when using the MEMORY command.

The other cause of this error is when the program or it s variables get too large for the available RAM. If this is the case, you will somehow have to cut down the size of the program; shorten variable names, use Integer variables wherever possible, use multi-statement lines.

A third cause is where a series of FOR...NEXT, WHILE...WENDS or GOSUB...RETURNS are very deeply nested. This can happen with tiny programs in memory, as we are troubled here by a loss of space for the information about the active control structures rather than memory for storing programs or variables.

A similar error is 'String Space Full', caused by there being too many strings in the memory. The setting up of strings in a computer is often a memory intensive task. Try the below:

```
10 PRINT FRE(0)
20 A$=STRING$(50,"#")
30  PRINT FRE(0)
40 A$=STRING$(70,"#")
50 PRINT FRE(0)
60 A$="HELLO"
70 PRINT FRE(0)
```

Running this program on a CPC 6128 returned the following figures:

```
42144
42085
42013
42013
```

Comparing these figures with the program will reveal that although A$ ended up with only five characters in it ("HELLO"), the system had still allocated to it room for a longer string. This is how the string space get s filled up. The extra memory, that at one time was used for string storage but is now not needed is not available for general use. A process called 'Garbage Collection' is carried out by BASIC on occasion to reclaim this memory. We can force such an event whenever we need it by using:

```
PRINT FRE("")
```

rather than PRINT FRE(0). If we replace each occurrence of FRE(0) in the above routine with FRE("") then my system returned:

```
42140
42081
42061
42133
```

This obviously has returned more memory to us through the garbage collection process. The way to avoid this error, therefore, is to do a garbage collection at frequent intervals when a lot of string processing is going on. The result of FRE("") doesn't have to be printed; it can just as easily be assigned to a variable. One possibility would be to attach a line such as:

```
dummy=FRE(" ")
```

to a routine that is called by an EVERY statement.

Numeric Errors

There are a wide range of errors caused by numeric problems, such as 'Overflow', 'Divide by Zero', and so on. There isn't much you can do about these except try to make sure that they don't happen! They are usually generated by a user typing in some data that is outside the range expected by the program. This is where data validation comes in useful. 'Subscript out of range' can also be generated by invalid inputs.

Other numeric problems, such as 'Operand Missing' or 'Type Mismatch' are caused by programming errors.

Control Errors

There are several error conditions that can arise when the computer comes across part of a control structure, such as NEXT or WEND, when it's not expecting it. Such errors are 'Unexpected NEXT', 'Unexpected WEND', 'Unexpected RESUME', 'WEND Missing', 'NEXT Missing', and so on. These are commonly caused by programming errors or injudicious use of the GOTO statement. A useful habit to cultivate to get rid of some of these errors is to indent nested loops as you type them in by adding spaces after the line number. For example:

```
10 FOR J=0 TO 10
20 FOR I=0 TO 5
30 PRINT I*J
40 NEXT I
50 NEXT J
```

can also be typed in as:

```
10 FOR J=0 TO 10
20   FOR I=0 TO 5
30   PRINT I*J
40   NEXT I
50 NEXT J
```

which is clearer, although it uses more memory. All lines at the same level of 'nesting' are indented by the same number of spaces. Missing NEXTs or WENDs are thus clearly visible. This is another good reason for putting the control variable after the NEXT:

NEXT I rather than NEXT

because if the control variable isn't the one being used in the corresponding FOR loop then it's clear that a NEXT is missing, and the BASIC interpreter will signal this fact with an error.

Finding the Bug

Once an error has been flagged, and the line number obtained, it's simply a matter of editing the offending line. This is usually easy, but in multi-statement lines can be a little more difficult. The best solution here is to put each statement in such a line on a different line then re-run the program. This will enable you to get the offending statement.

This assumes that the error is a straightforward one; occasionally the error could well have occurred in a totally different part of the program to that in which the error is flagged. In such a case, TRACE can be useful, though screen fulls of line numbers leading up to the point at which the problem occurred can be confusing rather than helpful. Use the TRON and TROFF statements in program lines to turn the Trace facility on and off at the required places in the program. Check the values of variables in the offending statement against their expected values. If all else fails, go through the listing, playing at being a computer! This can be tedious, but often works.

Of course, the bug could be caused by your logic in designing the program, rather than the computer's execution of the program. So, check the algorithm that you used. Go through it with pencil and paper to see if it returns the correct results with input data that you have already got answers for. Common problems of this type are as follows.

1. Priority

Expressions passed to the Amstrad are evaluated on a priority basis, that we've already seen. If you want to get around this in built priority of arithmetical operations, say to do an addition before a multiplication is carried out, then the addition must be bracketed. So:

1+a*3 would be written as (1+a)*3

You will soon get used to this, but I tend to put more parentheses in than are absolutely necessary; it gives me peace of mind!

Use of AND and OR

The logical operators often cause problems, mainly because in spoken English we don't often exhibit the logical precision that the computer requires! The only solution here is to go through logical expressions carefully. In addition, remember that they can often be written in a couple of different ways. So,

```
IF a=6 AND b=7 THEN GOTO 2300
```

would require both a=6 and b=7 before line 2300 would be visited. The same job could be done by:

```
IF NOT(a<>6 OR b<>7) THEN GOTO 2300
```

The first version is much more obvious, and it's a good idea to try and simplify any logical expressions that you use.

Some rather hard to find bugs can be introduced if you are using AFTER, EVERY or similar interrupt driven routines. It is essential that you get these working BEFORE you try and debug a program containing them. I would suggest that these routines be tested by simply deleting the EVERY or AFTER statement from the program, and insert, somewhere in the program where it will be regularly executed, a GOSUB to the routine in question. Bugs can then be sorted out with you knowing the exact place in the program from where the EVERY routine was called, and also knowing the state of variables, etc. when it was called. Two error conditions associated with Interrupt routines that can catch out the unwary (they caught me out, repeatedly!) are:

1. Variables being used in an EVERY routine that are also used in the main body of the program. It's clear that you must take a great deal of care with the variables used in EVERY routines to ensure that any alteration of variables is either desirable, e.g. changing the position of something on the screen, or unimportant.

2. Timing. If you call an EVERY routine too frequently, your program can appear to 'hang up'. This is due to the computer attempting to service the EVERY routine too quickly. As soon as one execution of the routine has been completed, the next one is being requested. The solution here is to speed up the execution of the EVERY routine or decrease the rate at which it is called!

Useful Subroutines

We'll begin with a couple of input routines, which allow users to type responses in to the computer without using the INPUT or INPUT LINE statements. You may ask; why bother? Indeed, if no one else uses your programs then you can probably trust yourself not to type in 'daft' responses to questions asked by the machine. However, for other uses, such an input routine will screen out at least some of the potential errors. Remember that it's easier to prevent errors happening in a program than to clear them up when they have happened. The first routine is called TEXTINP.

```
   10 MODE 1
   20 le%=10
   30 emf%=0
   40 xp%=10:yp%=10
   50 g$="abcdefghijklmnopqrstuvwxyz":a$=g$+UPPER$(g$)
   60 GOSUB 30000
   65 PRINT
   70 PRINT s$
   80 END
   90 :
30000 REM a text input routine
30010 REM for use in programs.
30020 REM xp%, yp% is the position of
30030 REM the first character, a$ contains
30040 REM the acceptable characters, le%
30050 REM the maximum number of characters
30060 REM the string is returned in s$
30065 REM emf%=0 means empty strings not acceptable
30066 REM emf%=1 means empty strings are acceptable
30070 :
30080 IF LEN(a$)<1 THEN ERROR 5
30090 IF le%<1 THEN ERROR 5
30100 :
30110 REM ensure that length and a$ are
30120 REM reasonable values.
30130 :
30135 a$=a$+CHR$(13)+CHR$(127)
30140 LOCATE xp%,yp%
30150 s$=""
30160 :
30170 g$=INKEY$:IF g$="" THEN GOTO 30170
30180 flag=0: IF INSTR(a$,g$)=0 THEN SOUND 1,50:GOTO 30170
```

```
30190 char%=ASC(g$)
30200 IF char%=127 AND s$="" THEN SOUND 1,50:GOTO 30170
30210 IF char%=127 THEN flag=1
30220 IF emf%=1 THEN IF (char%=13 AND s$="") THEN RETURN
30230 IF emf%=0 THEN IF (char%=13 AND s$<>"") THEN RETURN
30240 IF flag=0 THEN IF LEN(s$)<le% THEN s$=s$+g$:LOCATE
      xp%,yp%:PRINT s$; ELSE SOUND 1,50
30250 IF flag=1 THEN
   c$=MID$(s$,1,(LEN(s$)-1)):s$=c$:c$=LEFT$(c$+STRING$(le%,"
      "),le%):LOCATE xp%,yp%:PRINT c$;
30260 GOTO 30170
```

This routine is simple enough to use; xp% and yp% set the position of the input in text coordinates. The input will be in the current text pen colour. 'le%' is the maximum number of characters that is to be accepted, and emf% determines whether or not an 'empty string' is to be accepted as a response. (Obtained by pressing <ENTER> alone without any other characters being entered). emf%=0 will NOT accept such empty strings, while emf%=1 will. 'a$' holds all the characters that are to be accepted by the routine. In the demonstration part of the above listing, the routine has been programmed to accept all upper and lower case letters of the alphabet. In the relevant line of the program, I originally tried the line:

```
a$="abcdefghijklmnoqrstuvwxyz"+UPPER$(a$)
```

but it didn't seem to work. So, I replaced it with:

```
g$="abcdefghijklmnopqrstuvwxyz":a$=g$+UPPER$(g$)
```

The string typed in is terminated with the <ENTER> key, as with a normal INPUT statement. Any character that is not in 'a$' is not accepted in to the input string, which is in 's$' when the subroutine returns.

NUMINP

This routine serves a similar function as the above, but is dedicated to accepting numeric input.

```
10 MODE 1
20 ma=10:mi=1
40 xp%=10:yp%=10
50 le%=10
```

```
60 GOSUB 30000
65 PRINT
70 PRINT num
80 END
90 :
30000 REM a number input routine
30010 REM for use in programs.
30020 REM xp%, yp% is the position of
30030 REM the first character, ma is the
30040 REM largest number available and
30050 REM mi is the minimum number acceptable
30060 REM the value is returned in num
30065 REM le% is the max. number of characters
30066 REM acceptable.
30070 :
30080 a$="0123456789.-"
30090 IF ma<mi THEN ERROR 5
30100 :
30110 REM ensure that length and a$ are
30120 REM reasonable values.
30130 :
30135 a$=a$+CHR$(13)+CHR$(127)
30140 LOCATE xp%,yp%
30150 s$=""
30160 :
30170 g$=INKEY$:IF g$="" THEN GOTO 30170
30180 flag=0: IF INSTR(a$,g$)=0 THEN SOUND 1,50:GOTO 30170
30190 char%=ASC(g$)
30200 IF char%=127 AND s$="" THEN SOUND 1,50:GOTO 30170
30205 IF char%=127 AND s$<>"" THEN flag=1
30210 IF (char%=13 AND s$="") THEN GOTO 30170
30220 IF char%=13 AND VAL(s$+".0")>=mi THEN
      num=VAL(s$):RETURN
      30225 IF char%=13 AND VAL(s$+".0")<mi THEN
      s$="":LOCATE xp%,yp%:PRINT STRING$(le%," ");:SOUND
      1,50:GOTO 30170
30240 IF flag=0 AND (VAL(s$+g$+".0")<=ma AND LEN(s$)<le%)
      THEN s$=s$+g$:LOCATE xp%,yp%:PRINT s$; ELSE SOUND
      1,50
30250 IF flag=1 THEN
      c$=MID$(s$,1,(LEN(s$)-1)):s$=c$:c$=LEFT$(c$+STRING$
      (le%," "),le%):LOCATE xp%,yp%:PRINT c$;
30260 GOTO 30170
```

```
20351 FOR row%=1 TO 8
20352 l%(row%)=VAL("&X"+b$(row%))
20354 NEXT
20360 RETURN
20370 :
20400 REM sideways routine 2
20405 FOR g%=1 TO 8:b$(g%)="":NEXT
20410 FOR row%=1 TO 8
20420 c$=BIN$((PEEK(35999+row%)),8)
20520 FOR column%=8 TO 1 STEP -1
20530 b$(column%)=b$(column%)+MID$(c$,column%,1)
20540 NEXT:NEXT:d%=8
20550 FOR row%=1 TO 8
20560 l%(row%)=VAL("&X"+b$(d%)):d%=d%-1
20570 NEXT
20580 RETURN
```

xp%,yp% and le% have the same function as they did in TEXTINP. 'mi' is the minimum value that will be accepted, and 'ma' is the maximum value that will be accepted. Integers and numbers containing decimal points will be accepted, as will negative numbers.

A digit will NOT be accepted if accepting that digit would take the value of the number over 'ma'. On pressing <ENTER>, the number is checked to see if it is greater than or equal to 'mi'. If it isn't, then the user is prompted to type in another number. The number will be returned in the variable 'num', and the string of characters that generate it will be returned in 's$'.

There are a variety of alterations that can be made to these two simple routines. For example, the 'box' drawing subroutines seen previously could be used to put a box around the area of the screen in which the input is to be made. In addition, the double height subroutine could be incorporated into the above routines to give input routines that print the inputted characters larger than usual.

Sort Routines

Once data is in the computer, stored in arrays, etc. it's often useful to be able to sort it into some sort of order; numeric or alphabetic, depending upon whether you're dealing with numbers or strings. Locomotive BASIC already includes commands to obtain the largest and smallest numbers in a list; here we'll see a simple routine to sort arrays into order. The programs will sort integer arrays, but it is an easy job to get them to sort string or real arrays simply by altering the type of arrays used in the subroutines. Subroutines to perform sort

operations in BASIC have one disadvantage, though; they are often fairly slow. However, machine code routines can easily be written to sort string and integer arrays into the correct order fairly quickly using the information presented in Chapters 1 and 2 of this book.

The Bubble Sort

There are a variety of sort routines available. This is one that is reasonably efficient in most circumstances for smallish quantities of data. The order into which the data is sorted depends on the operator in line 20090 of 'BUBSORT':

> sorts the numbers into descending order.
< sorts the numbers into ascending order.

```
   10 REM Bubble Sort Demonstration
   20 REM Joe Pritchard, Nov. 1985
   30 :
   31 MODE 2
   32 PRINT "Raw Data":t=TIME
   35 num%=10
   40 GOSUB 1000
   50 GOSUB 20000
   60 GOSUB 2000
   70 END
   80 :
 1000 REM initialise array with random
 1010 REM numbers and list them out
 1020 DIM a%(30)
 1030 FOR i=1 TO num%
 1040 a%(i)=RND*100
 1050 PRINT a%(i)
 1060 NEXT i
 1070 RETURN
 1080 :
 2000 REM print out sorted array
 2005 PRINT "Sorted Data  ",TIME-t
 2010 FOR i=1 TO num%
 2020 PRINT a%(i)
 2030 NEXT
 2040 RETURN
 2050 :
20000 REM sort routine
20010 REM a%() holds an array of numbers
```

```
20020 REM to be sorted out into order
20030 REM uses Bubble Sort algorithm
20040 REM enter routine with num%
20050 REM holding number or entries
20060 :
20070 i%=0
20075 REM replaces outer loop
20080  FOR j%=2 TO num%
20090,IFa%(j%)<a%(j%-1)THEN
      temp%=a%(j%-1):a%(j%-1)=a%(j%):a%(j%)=temp%:i%=j%-1
20100  NEXT j%
20110 IF i%>1 THEN GOTO 20070
20120 RETURN
```

The operation of this routine is fairly simple. Repeated comparisons are made of each item in the array with the item following it. This is done by the inner of the two FOR...NEXT loops. This will result in each pair of items being sorted into the correct order. This is then repeated num%-1 times by the outer FOR...NEXT loop, where num% is the *number of items to be sorted in the array.*

This will eventually sort ALL the items in to the correct order. As it stands, this isn't very efficient, as the outer loop will repeat num%-1 times even if the items in the array have been already sorted by earlier passes. A 'flag' can be introduced into the program to detect this occurrence, and this makes the program more efficient in some circumstances, thus making it take less time. This modification has been added to the program 'BUBSORT2'.

```
  10 REM Bubble Sort Demonstration
  20 REM Joe Pritchard, Nov. 1985
  30 :
  31 MODE 2
  32 PRINT "Raw Data":t=TIME
  35 num%=10
  40 GOSUB 1000
  50 GOSUB 20000
  60 GOSUB 2000
  70 END
  80 :
1000 REM initialise array with random
1010 REM numbers and list them out
1020 DIM a%(30)
1030 FOR i=1 TO num%
1040 a%(i)=RND*100
```

```
 1050 PRINT a%(i)
 1060 NEXT i
 1070 RETURN
 1080 :
 2000 REM print out sorted array
 2005 PRINT "Sorted Data  ",TIME-t
 2010 FOR i=1 TO num%
 2020 PRINT a%(i)
 2030 NEXT
 2040 RETURN
 2050 :
20000 REM sort routine
20010 REM a%() holds an array of numbers
20020 REM to be sorted out into order
20030 REM uses Bubble Sort algorithm
20040 REM enter routine with num%
20050 REM holding number or entries
20060 :
20070 i%=0
20075 REM replaces outer loop
20080  FOR j%=2 TO num%
20090,IFa%(j%)<a%(j%-1)THEN
      temp%=a%(j%-1):a%(j%-1)=a%(j%):a%(j%)=temp%:i%=j%-1
20100  NEXT j%
20110 IF i%>1 THEN GOTO 20070
20120 RETURN
```

Here we dispense with the outer FOR...NEXT loop and the number of times that the outer loop is executed is determined by the state of the data.

The time taken to execute a Bubble sort of either of the above types depends upon the state of the 'raw' data presented to the routine and the number of elements to be sorted. For BUBSORT time is proportional to num%*num% where num% is the number of items in the array. If the data is already partially sorted, then BUBSORT2 will take less time than BUBSORT.

You may ask why we bother sorting data in this fashion. One reason is when we are searching through a table of data, say names, to find a particular piece of information. Say we wish to find the name 'Pritchard' in a string array. There are a few approaches.

Linear Search

The Linear Search is very simple, very crude, and can be slow. The data being searched is not sorted into order, and we start at array

element 1 and work through to the end of the string array. This can take a long time if the 'Pritchard' entry is at the end of the array, or if it's not in at all!

Alphabetical Search

Here, the array is first sorted into alphabetical order. We then search from element 1 until the first element whose first letter is 'greater than' 'P'. If we haven't found 'Pritchard' by this time he's not in the list. This will speed up the search time in many instances.

Binary Chop

No, not a new form of martial art ("Way of the Exploding Binary Chop"?) but a technique which uses a sorted list of data.

Say we have 100 items, sorted into alphabetical order. We examine item 50. If this is 'less than' 'Pritchard', then we know, due to the data being sorted, that 'Pritchard' cannot be in the first 50 entries. We now look at item 75. Again, if it's less than 'Pritchard' we know that 'Pritchard', if present, must be in the last twenty five entries. We carry on in this fashion until we have only a few entries to examine, or we're sure that 'Pritchard' isn't in the list. We can then examine the remaining entries in turn until we find 'Pritchard'. This technique cuts down the number of comparisons we have to make and hence speeds up the process.

If we have a simple database in which data is stored in several arrays, say name and address in one array and telephone number in another, then when we sort the data into order we must sort EACH array into order using ONE array as the one to be sorted. We say that we are sorting on a particular array. For example:

Names	Age
Joe	24
Fred	32
Neil	47
Billy	2

sorted on 'Age' we'd get:

Billy	2
Joe	24
Fred	32
Neil	47

but sorted on 'Name' we might get:

Billy	2
Fred	32
Joe	24
Neil	47

Failure to sort all the arrays into order on the same field can lead to a phenomenon known in the computer trade as a 'mess'!

The whole business of manipulating data in this way is a vast subject. We'll leave it there.

I'd like to finish this Chapter, and the book, with brief looks at the aspects of the Amstrad 464 system that I haven't looked at yet; the cassette and keyboard.

The Keyboard

Let's begin with a look at the Firmware routines that we can use directly from BASIC by simply CALLing the address given here.

CALL &BB18.

I have used this in some of the BASIC routines listed in this book, and it simply causes the machine to wait until a key is pressed. The user gets no indication of what the key was that was pressed, but it is a useful call.

CALL &BB00.

This call is of less use but is still a handy call to know. It is the Key Manager Initialise routine, which resets the Key Manager part of the Firmware. This is the section of the Firmware that looks after the keyboard. For the BASIC programmer the implications are as follows.

i/ Key Expansions are set back to their default values.
ii/ Key translations are set back to their 'turn on' values.
iii/ Repeat Speed and the keys that can Repeat are set back to their initial values.
iv/ Shift and Caps lock are turned off, and the Break Event is disarmed.

The call thus offers a quick way of clearing the keyboard.

The other Key Manager routines are not useful from BASIC, as they need setting up with parameters in the Z-80 registers before they can be useful. The behaviour of the keyboard can, however, be modified by BASIC commands. I will give a quick review of these here.

Key Repeat

Basic commands allow us to alter the rate of key repeat AND the keys that are allowed to repeat. Starting with repeat rate, the SPEED KEY command is used as following:

```
SPEED KEY delay,rate
```

'delay' specifies the delay in fiftieths of a second before key repeat starts and 'speed' specifies the time between each subsequent repeat, again in fiftieths of a second. The default values are 30 and 2 respectively. When programming, it's a good idea to have some means of returning the keyboard repeat parameters back to their normal values for when you wish to edit the program. Typing in anything can be impossible with parameters of, for example, 1,2! I usually have a function key programmed with the appropriate SPEED KEY command when developing a program, but in a running program I usually have the following pieces of code in the program.

1. Any ON ERROR trap set up contains a SPEED KEY command to set the repeat rates appropriately in case the user has to type in some information before normal program execution is resumed. Before going back to the main part of the program, the key repeat rate can be set to the value required by the rest of the program.

2. An option to leave the program should leave the machine with the keyboard repeat rates set to their default value.

In programs to be used by non-computer programmers, fingers tend to be left on keys for a long time; this can cause unwanted repeats. The way around this is twofold;

1. Turn off the repeat on some keys altogether, as is done by the Firmware with the ENTER key.

2. Make the 'delay' parameter in the above as long as possible, so that repeats are only given when the user deliberately holds the key down for a length of time.

Note that all keys able to repeat repeat at the same rate.

To turn off the repeat facility for a given key we use the KEY DEF command:

```
KEY DEF number,repeat
```

where 'number' is the Key Number of the key in question. This is NOT the ASCII code returned by the key; it is the 'reference number' used by the hardware for a particular key. The key number for a given key can be found in the Manual for 464/664 users and on the computer itself for 6128 users. A 'repeat' parameter of 0 turns off the repeat for that key, and a '1' turns it on.

This can be particularly useful for programs in which several screens of text are to be viewed by the user pressing the space bar, etc. to go on to the next page. Here, turn off the repeat on the Space bar to prevent the user 'running through' all the screens by inadvertently leaving his finger on the key!

KEY DEF

This is a very useful command indeed. It allows us to alter the ASCII code returned by a particular key when that key is pressed. We've already used the simplest form of it to control whether or not a key repeats. The full syntax is:

```
KEY DEF number,repeat,normal, shift,ctrl
```

where 'normal' is the ASCII code to be returned when the key is pressed on it s own, 'shift' when shift is pressed and 'ctrl' when the control key is pressed at the same time as the key. Try the below line to see it in action:

```
KEY DEF 47,1,32,65,66
```

which programs the space bar (key number 47), to return 32 when pressed alone, ASCII code 65 ('A') when pressed with shift and 66 ('B') when pressed with control. The letters entered by pressing SHIFT-SPACE or CTRL-SPACE are received by the computer as if they'd come from the A or B keys. This command makes data validation a little easier; we can program all the keys to return, say, their upper case values no matter what the SHIFT,CAPS-LOCK or CTRL status is at the time.

In addition, the command is useful in games programming, where we use the keyboard to control the game. Usually we have to use one key for each direction. Say we had a simple 'invaders' style game. We might consider using 'Z' to move left, 'X' to move right and 'space' to fire. Well, this works but we occasionally lose our fingering on the keys and are zapped while we get back to the correct keys. However, with KEY DEF, you could define all the keys on the left hand side of the keyboard to return the ASCII code for 'Z' and all those on the right hand side to return the ASCII code for 'X'. Apart from increasing your high score, it spreads the wear and tear across more keys on the keyboard! The ASCII codes given to keys using the KEY DEF command can be the codes for Expansion Tokens, in the range 128 to 159, thus allowing normal keys, say CTRL-A onwards, to return strings that have been programmed into the 'function keys'.

The final keyboard interrogation command we'll examine is INKEY. In this book we've used INKEY$. Whereas that command returned the character associated with a given key, the INKEY command uses Key Numbers, and tells us whether a particular key is pressed or not. It is thus of use in games, or while waiting for a key to be pressed. The syntax is:

```
INKEY(number)
```

where the value of 'number' is the Key Number of interest.

Thus if we were interested in the status of the Space Bar we would use 47 as the value of number, as this is the Key Number for the space bar. The value returned by this function depends upon the status of Shift and CTRL as well as whether or not the key in question is pressed. If the key in question isn't pressed then a value of -1 is returned, irrespective of the status of Shift and CTRL. Thus, to wait for the Space Bar to be pressed we could have:

```
1000 key=INKEY(47)
1010 IF key=-1 THEN GOTO 1000
```

If neither Shift or CTRL is pressed then a value of 0 will be returned. CTRL pressed will set bit 7 of the value returned to 1 and Shift will set bit 5 to 1 if pressed. If both are pressed, both bits will be set to 1.

One potential problem with this call is that the key pressed is NOT removed from the keyboard buffer, and is still there, waiting for an INPUT or INKEY$ to remove it. The safest policy here is to get rid of the key press before going further. One method, which can be used on

any Amstrad, is add the following line to the above routine:

```
1020 a$=INKEY$
```

This will set a$ to hold, in this case, " " when the space bar is pressed. On systems using Locomotive BASIC 1.1, line 1020 can be replaced with:

```
1020 CLEAR INPUT
```

This will flush the character from the input buffer. Note that INKEY looks at the keyboard every fiftieth of a second.

The Cassette System

There isn't much we can do with the Cassette system from BASIC except use it; I gave some useful machine code routines to make use of the cassette interface in my book 'Ready Made Machine Language Routines for your Amstrad', also published by Melbourne House. Again, let's begin with an examination of the Firmware routines that we can CALL directly from BASIC.

CALL &BC65

This resets the Cassette Manager portion of the Firmware, closes files and sets up the default rate for tape write operations.

CALL &BC6E

This closes the Cassette Motor Control Relay, and will hence cause the cassette motor to start if PLAY or RECORD are pressed.

CALL &BC71

This opens the Cassette Motor Control Relay and so stops the cassette motor if it's running.

CALL &BC7D

This aborts any cassette read operation in progress. A call to &BC92 does the same job for any write operation in progress.

Apart from the BASIC save and load commands, the only command that alters the status of the cassette system is the SPEED WRITE command, which sets the speed at which data is written to tape.

```
SPEED WRITE 0
```

is the default rate of 1000 baud and is generally more reliable.

SPEED WRITE 1

will send data out at 2000 baud. Faster, but on the whole less reliable. In practice, I find that both baud rates work fine but it's more important to keep your tape drive clean for the higher baud rate. 6128 or 664 users may be best advised to keep to 1000 baud due to the fact that the tape recorders used with these machines may not be as good at the job as the integral one built in to the 464.

The software for reading automatically determines the write rate used on the file being read in from the File Header Leader tone. A cassette operation can generate error messages where appropriate. 'Read Error a' or 'Read Error b' are generated when the machine has difficulty reading data from the cassette tape. This can be caused by dirty tape heads or pinch roller in any system, or by not setting the volume and tone controls of an external tape recorder properly for 664 or 6128 systems. In addition, problems can be experienced by loading a tape recorded on one tape recorder into a machine using another tape recorder. This problem isn't important between different 464 computers as for all intents and purposes the recorders are identical. It is more likely to cause problems on 664/6128 systems or 464 machines running tapes recorded using different recorders. There's not much you can do here, but follow the below simple rules for saving and loading. Apart from the 'cleanliness' rules, they aren't necessary for 464 systems.

Cleanliness

Clean the recording heads in the tape recorder fairly frequently, using cotton buds and a tape head cleaning solution. This is usually based on Ethanol or Propanol, with a touch of detergent in it. Under NO circumstances should any water based cleaning solution be used. As soon as a cotton bud shows any sign of discolouration, it should be discarded.

Finish off the cleaning operation with a clean, dry cotton bud. Metal items, such as screwdrivers, should NEVER be used to clean tape heads.

In addition to the heads the 'Pinch Roller' should be cleaned. This is the rubber roller to one side of the cassette drive. This is best cleaned with one of the 'cleaning tapes' that has a series of felt pads which can be soaked in tape head cleaner. This 'tape' is then put in the drive and PLAY and RECORD are pressed. This causes the felt pads to 'brush' at the heads and Pinch Roller, thus removing the dirt. A

dirty pinch roller is particularly nasty; apart from causing variations in tape speed which can cause read errors, a really dirty one can crease the tape and so permanently damage it. This can be rather unpleasant, given the cost of software which, on the whole, cannot be backed up.

There are two other ways in which the performance of your cassette drive can be 'perked up'. The first is to have it 'aligned'. This is simply a procedure which maximises the signal that the tape head reads from the tape. You can get commercial kits which include instructions on how to do this. The second thing to consider might be the demagnetisation of the heads. You can get devices to do this. However, it should be considered only after you've done everything else. The tape heads can become permanently magnetised due to the magnetic tapes being moved past them.

This is a very slow process, however, and tape problems are more likely to be due to dirt.

External Recorders

The 664 and 6128 machines require that an external tape recorder be used when tape programs are to be run. This leads to a whole host of potential problems.

Positioning. The recorder shouldn't be too close to any source of magnetic fields.

Tape Lead. The lead shouldn't run parallel to any mains leads, under monitors, etc. or anywhere near magnetic or electric fields.

Batteries or Mains. If you MUST use batteries, then 'High Power' batteries are essential. If possible, get alkaline batteries, as these will provide longer life with less fluctuation in tape speed as the batteries fail. This is the main problem with batteries; as they fail, the speed of the tape varies. This leads to the pitch of the recorded tones that represent the program information varying, and thus the computer may not recognise the signals as data.

Battery recorders can be useful if you've got 'dirty' mains; that is, a source of mains electricity that has electrical noise on it, from things like electric drills, thermostats, central heating pumps, people switching things on and off,etc. Most houses experience this problem occasionally, but in some houses it seems to be a constant problem. However, it's only in extreme cases that batteries should be considered; the tape system can usually handle a little noise.

RFI. This stands for Radio Frequency Interference, and I mention it here 'cos I once had it from a CB transmitter nearby. Radio signals turn up in the tape system, and if strong enough can cause loss of data.

Volume and Tone Control Settings

These are the most common cause of data loss. The volume must be high enough for the computer to 'hear', but if too high can cause distortion. This will then lead the computer to 'mishear' the tones representing data.

The Tone control should be set at about three quarters to full, giving a clear sound.

The settings for your machine will have to be found by trial and error.

Another error, 'Read Error d', is possible but would require that the data written to the tape has been written incorrectly. Error d indicates a 'Block too long' error. Even if you can read the tape data in, there are still potential problems to deal with. The following are those that have given me the most 'fun' while programming the Amstrad.

Memory Full

This error message is occasionally generated when an attempt is made to load in a binary file with HIMEM set to a value HER than the start address of the binary file. The solution to this is to set HIMEN to an appropriate value before trying to load the byte file in. It is NOT usually possible to load a binary file representing a machine code program to a different address as the subroutine calls, absolute jumps etc, within the machine code with be incorrect for the new address.

SYMBOL AFTER

This command cannot be used whilst a file is opened (file opened by OPENOUT or OPENIN) unless a previous SYMBOL AFTER 256 had been issued. This is due to the fact that the act of opening a file to tape sets up a 2k buffer. SYMBOL AFTER needs a continuous block of memory around HIMEM for subsequent character definitions, and with the file buffer in the way it cannot do this. The error 'Improper Argument' is returned.

Other Problems

With large programs, or with programs that have brought HIMEM down to a few thousand or so, any attempt to open a file will produce the 'Memory Full' message; there isn't enough room for the Firmware to set the 2k buffer up.

Once a tape operation has started the only way out before it's finished is to use Escape. Escape is checked for directly; the key press is not put in to the keyboard buffer. The practical result of this is that you occasionally have to hold the Escape key down for a length of time, until the keyboard is scanned again.

Appendix 1
ROM Routines Used

The BASIC ROM routines are not publicised in any way by Amsoft, as they do not wish to encourage the development of software which may not run on all machines that they manufacture. However, in this book I have used two BASIC ROM routines. The addresses of the routine in BASIC 1.0 and BASIC 1.1 are shown below.

Error Entry Point:
BASIC 1.0 &CA93
BASIC 1.1 &CB55

Break Event Handler:
BASIC 1.0 &C45E
BASIC 1.1 &C492

These routines are used in Chapter 2, and so users of BASIC version 1.1 should use the address given above rather than the one in the program listings in Chapter 2.

Appendix 2
BASIC Tokens

This information should be used in conjunction with that from Chapter 1, and is useful if you wish to examine the structure of a BASIC program.

&00	ABS	&01	ASC	&02	ATN
&03	CHR$	&04	CINT	&05	COS
&06	CREAL	&07	EXP	&08	FIX
&09	FRE	&0A	INKEY	&0B	INP
&0C	INT	&0D	JOY	&0E	LEN
&0F	LOG	&10	LOG10	&11	LOWER$
&12	PEEK	&13	REMAIN	&14	SGN
&15	SIN	&16	SPACE$	&17	SP
&18	SQR	&19	STR$	&1A	TAN
&1B	UNT	&1C	UPPER$	&1D	VAL
&1E	-	&40	EOF	&41	ERR
&42	HIMEM	&43	INKEY$	&44	PI
&45	RND	&46	TIME	&47	XPOS
&48	YPOX	&71	BIN$	&72	DEC$
&73	HEX$	&74	INSTR	&75	LEFT$
&76	MAX	&77	MIN	&78	POS
&79	RIGHT$	&7A	ROUND	&7B	STRING$
&7C	TEST	&7D	TESTR	&7E	-
&7F	VPOS	&80	AFTER	&81	AUTO
&82	BORDER	&83	CALL	&84	CAT
&85	CHAIN	&86	CLEAR	&87	CLG
&88	CLOSEIN	&89	CLOSEOUT	&8A	CLS
&8B	CONT	&8C	DATA	&8D	DEF
&8E	DEFINT	&8F	DEFREAL	&90	DEFSTR
&91	DEG	&92	DELETE	&93	DIM
&94	DRAW	&95	DRAWR	&96	EDIT

&97	ELSE	&98	END	&99	ENT
&9A	ENV	&9B	ERASE	&9C	ERROR
&9D	EVERY	&9E	FOR	&9F	GOSUB
&A0	GOTO	&A1	IF	&A2	INC
&A3	INPUT	&A4	KEY	&A5	LET
&A6	LINE	&A7	LIST	&A8	LOAD
&A9	LOCATE	&AA	MEMORY	&AB	MERGE
&AC	MID$	&AD	MODE	&AE	MOVE
&AF	MOVER	&B0	NEXT	&B1	NEW
&B2	ON	&B3	ON BREAK	&B4	O N ERROR GOTO
&B5	ON SQ	&B6	OPENIN	&B7	OPENOUT
&B8	ORIGIN	&B9	OUT	&BA	PAPER
&BB	PEN	&BC	PLOT	&BD	PLOTR
&BE	POKE	&BF	PRINT	&C0	-
&C1	RAD	&C2	RANDOMISE	&C3	READ
&C4	RELEASE	&C5	REM	&C6	RENUM
&C7	RESTORE	&C8	RESUME	&C9	RETURN
&CA	RUN	&CB	SAVE	&CC	SOUND
&CD	SPEED	&CE	STOP	&CF	SYMBOL
&D0	TAG	&D1	TAGOFF	&D2	TROFF
&D3	TRON	&D4	WAIT	&D5	WEND
&D6	WHILE	&D7	WIDTH	&D8	WINDOW
&D9	WRITE	&DA	ZONE	&DB	DI
&DC	EI	&E3	ERL	&E4	FN
&E5	SPC	&E6	STEP	&E7	SWAP
&EA	TAB	&EB	THEN	&EC	TO
&ED	USING	&EF	=	&F1	<
&F4	+	&F5	MINUS	&F6	*
&F7	/	&F8	-	&F9	DIV
&FA	AND	&FB	MOD	&FC	OR
&FD	XOR	&FE	NOT		

In addition, the Version 1.1 BASIC will probably have additional tokens for the extra commands that are added to the BASIC.

Index

A
After....199-200
||AMPLITUDE....160
Animation....49-55

B
BASIC....111 et seq
Boxes....21-23
Break, effect on After....200
||BRKOFF....159
||BRKON....159

C
Cassette use....234-238
||CAT....159
Channel Synchronisation....99-81
Circles....32-34
Copying Screen Sections....46-49
||CRTC....159
CRTC....179
CRTC Registers....184-189
Curves....35-38

D
Debugging Programs....217-22
DI....200
Double Width Characters....7-8
DRAW program....68-71
||DUMP....161

E
Echo....95
EI....200
Ellipses....34-35
Envelopes....86-94
Errors in m/c programs....164-66
Eval....120-129
Event Blocks....166-168
EVERY....201-2

F
Filling in....38-46
||FIND....160
Flush....82
Frame Flyback....183

G
||GET....159
Glissando....99
Graphics....1 et seq
Graphics Screen....19-23

I
IF . . .THEN....120
||INK....160
Ink Modes....51-3
Input to Program....215-16
Interlace....182-3

J
Jumpblocks....161-164

K
Keyboard use....230-234

L
Large Characters....6
Line Length....111-2

Line Number`..............................111-2
||LLIST..160

M
Machine Code and BASIC.......161-178
Machine Code Loader............174-178
Mode 0..52
Mode 1..52
Mode 2..52
Modulation....................................94
||MOTOR......................................159
Musical Instruments....................96-98
Multicolour Characters......................17

N
NUMINP program....................223-225

O
ON ERROR..............................216-17
ON KEY................................169-173
ON SQ..84

P
Pallette Switching.......................55-68
Patching Jumpblocks...............161-62
||PAUSE......................................159
Pie Charts.................................74-76
Plane Graphics...............................39
PLAY program.......................104-108
Polygons..................................30-32
PRINT....................................119-20
Program Design.....................209-216
Program Size...............................139
Program Speed........................137-39
Program Storage......................111-12
Program Structure........................217
Program Style..........................209-16
Programmable Sound
Generator............................99-100
PSG Registers..........................100-5
Proportional Spacing......................3-4

Q
Queues.....................................82-5

R
REMAIN.......................................200
||RESET..................................158-9
Resolution....................................1-2
RHYTHM program.....................108-10
||ROMREAD.................................161
RSX routines.................................141

S
Screen Addresses.....................192-4
Screen Columns........................192-4
Screen Memory........................189-91
Screen Rows............................192-4
||SEARCH....................................160
Shadow effects..............................4-6
Sort routines...........................225-28
||SOUND.....................................159
SOUND command..........................77
||SPLITOFF..............................163-4
||SPLITON...............................163-4
Sprites.......................................62-8
SQ...82
Statement Storage........................112
||SYMBOL...................................159
SYMBOL..................................12-16
Sync Pulses.................................182

T
TEXTINP program.....................222-23
Text Printing...............................2-17
||TONE..160
Transformations.......................232-38
Two Dimensional Graphics................18

U
Upside Down Characters..................10
User Defined Characters.............12-13

V
Variable Assignments
Numeric...............................114-18
String.................................118-19
Variable Size Characters...............9-10
Variable Structure.....................129-37
Vibrato..98-9

W
Waveform..................................93-4

X
XOR graphics..............................52-5

NOTES

NOTES

NOTES

NOTES

Do you want to get just that little bit more out of the Amstrad BASIC interpreter? Joe Pritchard's latest book is an excellent collection of hints, tricks and routines that every Amstrad owner will want to read.

Locomotive BASIC is a powerful programming tool, and Joe Pritchard shows how to get the best out of it, as well as an introduction to expanding its capabilities with commands of your own.

Chapters covering graphics and animation, sound design and playback, and the way your BASIC program looks to the machine will enhance your programming style. RSX routines and short machine-code routine will expand the capabilities of your machine and make programming in BASIC more simple.

A detailed look at the screen is also provided, so you will be able to take complete screen control.

In short, this book is a must for any Amstrad CPC owner.

£12.95

Melbourne House Publishers

ISBN 0-86161-202-7

ADVANCED AMSTRAD BASIC

Joe Pritchard

Document numérisé avec amour par

AMSTRAD

CPC

MÉMOIRE ÉCRITE

https://acpc.me/