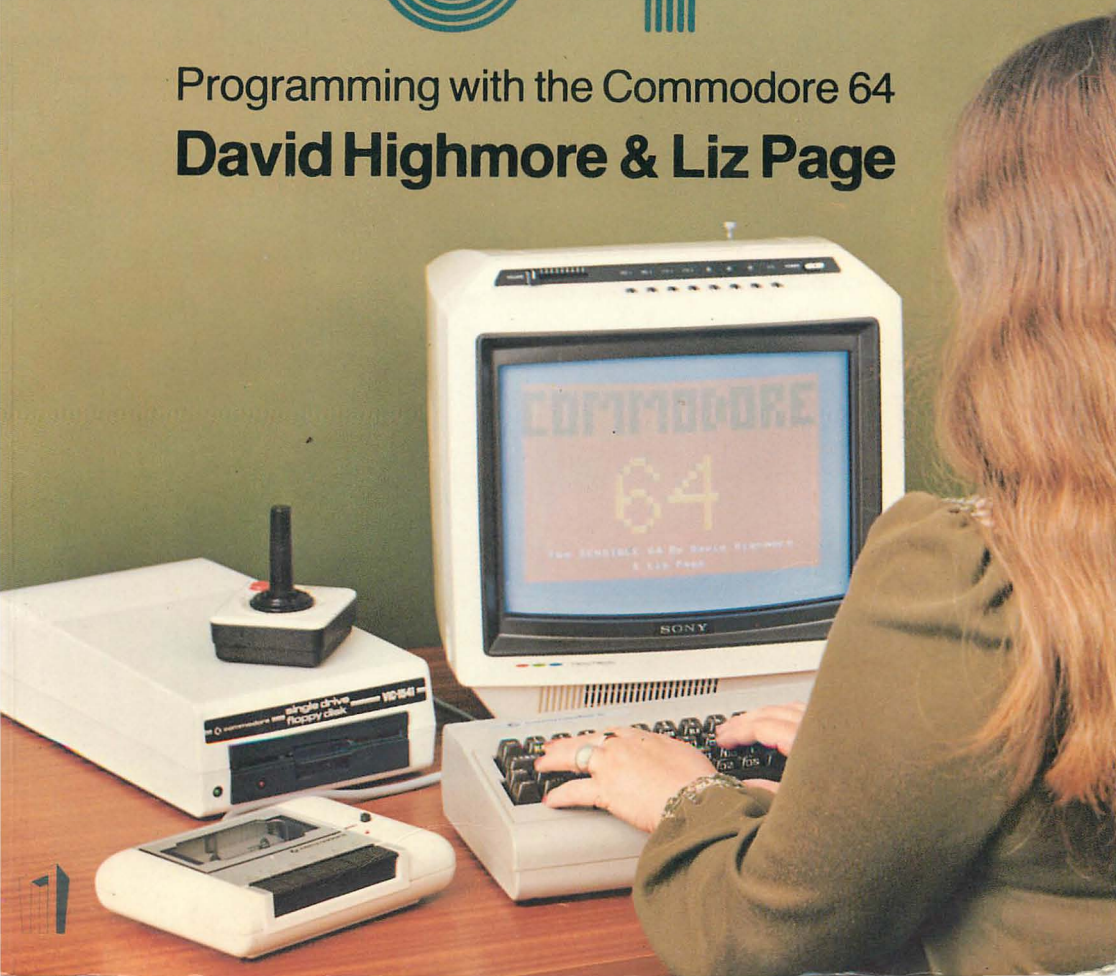
THE SENSIBLE 64

Programming with the Commodore 64

David Highmore & Liz Page



THE SENSIBLE 64

Programming with the Commodore 64

David Highmore & Liz Page

Micro books

First published in 1983 by
MICROBOOKS
443 Millbrook Road
Southampton
SO1 0HX

ISBN 0 946705 00 3

Printed by "OFFSET" Tel: Southampton (0703) 585544 (2 lines)

Sole U.K. Distributors:
John Wiley & Sons Ltd, Boffins Lane,
Chichester, Sussex.

The authors wish to express their sincere thanks to the following people:-

Brian & Julie Clatworthy, for the section on Music

Geoff Barber, Peter & Carol Green for helping to eliminate our mistakes,

and most of all to Beryl and Alan, without whom this book would never have been written.

CONTENTS

CONTENTS

INPUT TO THE COMPUTER

There are times when you may wish to access information from the keyboard without using the INPUT command. For example when you want to make use of the Function Keys or to ensure that only one key corresponding to a number is pressed.

There are two main ways of obtaining information from the keyboard. The first is the GET command which will take characters one at a time. Secondly you can PEEK the location 197.

GET STATEMENT

The GET statement in Basic will remove one character at a time from the keyboard buffer and if no character is present ie no key pressed it will return an empty string.

In the following example it will wait for any key to be pressed.

```
10 REM ** WAIT FOR ANY KEY **
15 PRINT"PRESS ANY KEY"
20 GOSUB10000
25 PRINT"OK THANKYOU"
30 END
10000 POKE198,0
10005 GETA$:IFA$=""THEN10005
10010 POKE198,0:RETURN
```

Line 10000 ensures there is no character in the keyboard buffer.

Line 10005 waits until a key is pressed.

Line 10010 clears the keyboard buffer and returns to the main programme.

In this case no information is passed back to the programme and any key will cause the programme to return.

In the next example the computer will only accept a Y or an N key as in "Do you want another go Y/N?".

```
10 REM ** Y OR N KEY **
20 PRINT"PRESS Y OR N"
25 GOSUB10000
30 IFY=1THENPRINT"YOU PRESSED 'Y'":END
35 PRINT"YOU PRESSED 'N'":END
10000 POKE198,0
10005 GET A$:IF A$=""THEN 10005
10010 IF A$ = "Y" THEN Y = 1:GOTO10025
10015 IF A$<>"N"THEN10000
10020 Y=0
10025 POKE198,0
10030 RETURN
```

Line 10000 clears the keyboard buffer.

Line 10005 waits until a key is pressed.

Line 10010 checks if it is a Y - if it is it sets a 'flag' and jumps to 10025.

Line 10015 checks if it is not an N - if it is not it starts the process of waiting for a key again.

Line 10020 sets the flag to 0 (N key pressed).

Line 10025 clears the keyboard buffer.

Line 10030 returns to the main programme.

There are two things that make this programme an advance over the previous one. The computer will only accept a Y or N and the variable 'flag' is set to indicate to the main programme

which key was pressed.

This can be adapted to select any two keys simply by substituting them for Y and N in lines 10010 and 10015.

If you wish to wait for a key in a particular range to be pressed this is best achieved by comparing the ASCII code of the character against the value of the range required.

```
10 REM ** ONLY ALPHABET KEYS **
20 PRINT"PRESS ANY LETTER"
25 GOSUB10000
30 PRINT"OK. THANKYOU"
35 END
10000 POKE198,0
10005 GET A$:IF A$ = ""THEN 10005
10010 IFASC(A$)<65ORASC(A$)>90THEN 10000
10015 POKE198,0:RETURN
```

Line 10000 clears the keyboard buffer.

Line 10005 waits for a key to be pressed.

Line 10010 checks to see if it is a letter of the alphabet. If it is not it then waits for another key to be pressed.

Line 10015 clears the keyboard buffer and returns to the main programme.

```
10 REM ** NUMBERS ONLY **
15 PRINT"ENTER A NUMBER 0 TO 9 "
20 GOSUB10000
25 PRINT"OK. THANKYOU"
30 END
10000 POKE198,0
10005 GET A$:IF A$ = ""THEN 10005
10010 IFASC(A$)<48ORASC(A$)>57THEN 10000
10015 POKE198,0:RETURN
```

This is essentially the same as before except that it will wait until a number between 0 and 9 is pressed. This is useful if you wish to cause the programme to branch to for example one of five parts of a programme depending on which key is pressed. This is useful in choosing an option from a 'menu'.

```
10 REM ** CHOOSE LINE **
15 PRINT"ENTER NUMBER 1 TO 5"
20 GOTO10000
100 PRINT"YOU CHOSE NUMBER 1":END
200 PRINT"YOU CHOSE NUMBER 2":END
300 PRINT"YOU CHOSE NUMBER 3":END
400 PRINT"YOU CHOSE NUMBER 4":END
500 PRINT"YOU CHOSE NUMBER 5":END
10000 POKE198,0
10005 GETA$:IFA$=""THEN 10005
10010 IFASC(A$)<49ORASC(A$)>53THEN 10000
10015 A=VAL(A$)
10020 POKE198,0
10025 ON A GOTO 100,200,300,400,500
```

Line 10000 clears the keyboard buffer.

Line 10005 waits for a key to be pressed.

Line 10010 checks if it is in the range 1 to 5.

Line 10015 converts it into a numeric variable.

Line 10020 clears the keyboard buffer.

Line 10025 branches to lines 100 to 500 on the value of the key pressed.

The numbers can of course be selected to suit the individual needs of the programmer.

All these examples have used the GET statement. Another way to obtain data from the keyboard

is to look at the NUMBER of the actual key pressed. This will be found at location 197 ie PEEK(197). This will give the value of the key pressed.

```
12 REM ** PEEK FOR ANY KEY **
15 PRINT"PRESS ANY KEY"
20 GOSUB10000
25 PRINT"OK. THANKYOU"
30 END
10000 IFPEEK(197)=64THEN 10000
10005 POKE198,0:RETURN
```

Line 10000 waits until a key is pressed.

Line 10005 clears the keyboard buffer and returns to the programme.

It should be noted that if no key is pressed PEEK(197) will be 64.

```
10 REM ** PEEK SPACE BAR **
15 PRINT"PRESS SPACE BAR"
20 GOSUB10000
25 PRINT"OK. THANKYOU"
30 END
10000 IFPEEK(197)<>60THEN10000
10005 POKE198,0
10010 RETURN
```

This is an advance on the previous programme as it will wait until the space bar is pressed.

Line 10000 waits until PEEK(197) = 60 ie when the space bar is pressed.

Line 10005 clears the keyboard buffer.

Line 10010 returns to the programme.

```
10 REM ** PEEK RETURN **
15 PRINT"PRESS RETURN"
20 GOSUB 10000
25 PRINT"OK. THANKYOU"
30 END
10000 IFPEEK(197)<>1THEN 10000
10005 POKE198,0
10010 RETURN
```

This will wait until the RETURN key is pressed.

```
10 REM ** ANY FUNCTION KEY **
15 PRINT"PRESS A FUNCTION KEY"
20 GOSUB10000
25 PRINT"OK. THANKYOU"
30 END
10000 N=PEEK(197)
10005 IFN<30RN>6THEN 10000
10010 POKE198,0
10015 RETURN
```

Line 10000 sets variable N to PEEK(197).

Line 10005 checks to see if it is one of the Function Keys.

Line 10010 clears the keyboard buffer.

Line 10015 returns to the programme.

This will wait until any of the Function keys are pressed.

```
10 REM ** CHECK A FUNCTION KEY **
15 PRINT"PRESS FUNCTION KEY 1 <F-1>"
20 GOSUB10000
25 PRINT"THANKYOU F-1 PRESSED"
30 END
10000 IFPEEK(197)<>4THEN 10000
10002 REM
10010 REM ****   4 WAITS FOR F-1   ****
```

```
10015 REM ****   5 WAITS FOR F-3   ****
10020 REM ****   6 WAITS FOR F-5   ****
10025 REM ****   3 WAITS FOR F-7   ****
10027 REM
10030 POKE198,0
10035 RETURN
```

Line 10000 waits until a specific Function Key is pressed from the value shown in 10002 to 10027.

Line 10030 clears the keyboard buffer.

Line 10035 returns to the main programme.

The following programme will enable you to use the Function Keys to make decisions.

```
10 REM ** CHOOSE FROM F-KEY **
15 PRINT"PRESS A FUNCTION KEY"
20 GOTO10000
100 PRINT"FUNCTION KEY 1 PRESSED":END
200 PRINT"FUNCTION KEY 3 PRESSED":END
300 PRINT"FUNCTION KEY 5 PRESSED":END
400 PRINT"FUNCTION KEY 7 PRESSED":END
10000 N=PEEK(197)
10005 IF N < 3 OR N > 6 THEN 10000
10010 IF N = 4 THEN N = 1:GOTO10030
10015 IF N = 5 THEN N = 2:GOTO10030
10020 IF N = 6 THEN N = 3:GOTO10030
10025 IF N = 3 THEN N = 4
10030 POKE198,0
10035 ON N GOTO 100,200,300,400
```

Line 10000 sets variable N to PEEK(197).

Line 10000 sets variable N to PEEK(197).

Line 10005 checks it is a Function Key.

Lines 10010 to 10025 set N according to which key is pressed.

Line 10030 clears the keyboard buffer.

Line 10035 will cause the programme to branch to one of four lines according to the key pressed.

The line numbers can be changed to fit in with your programme and the range of keys that will be accepted is also variable.

This section has dealt with a number of means to enable you to have more control over the way in which information input to the computer is handled. By using a range of techniques it is possible to compel the computer to accept only what you want it to.

```
10 REM ** ENTER WORD LENGTH **
50 INPUT"□WORD LENGTH";L
1000 W$="":PRINT"□"
1005 FORN=1TOL
1010 PRINT"■ENTER WORD ";W$
1015 POKE198,0
1020 GETA$:IFA$=""THEN 1020
1025 W$=W$+A$
1030 NEXT
1035 PRINT"■WORD ENTERED IS ";W$
```

Line 50 inputs the length of the word required.

Line 1000 sets W$ to null.

Line 1005 sets FOR-NEXT loop to length required.

Line 1010 prints the word.

Line 1015 clears the keyboard buffer.

Line 1020 waits for the key to be pressed.

Line 1025 adds the character to W$.

Line 1030 repeats the process until the word is the correct length.

Line 1035 prints the final word.

This will enable you to input a word one letter at a time until the word is of the required length. By simply adding a few lines it is possible to force the computer to accept only alphabet characters or numbers or even both.

```
10 REM ** LIMIT TO ALPHABET **
50 INPUT"□WORD LENGTH";L
1000 W$="":PRINT"□"
1005 FORN=1TOL
1010 PRINT"■ENTER WORD ";W$
1015 POKE198,0
1020 GETA$:IFA$=""THEN 1020
1025 IFASC(A$)<65ORASC(A$)>90THEN 1015
1030 W$=W$+A$
1035 NEXT
1040 PRINT"■WORD ENTERED IS ";W$
```

This is essentially the same but line 1025 will check that each character is a letter of the alphabet.

The following programme will accept only number keys and converts the word into a variable that may be manipulated as a number.

```
10 REM ** LIMIT TO NUMBERS **
50 INPUT"□NUMBER LENGTH";L
1000 W$="":PRINT"□"
1005 FORN=1TOL
1010 PRINT"■ENTER NUMBER ";W$
1015 POKE198,0
1020 GETA$:IFA$=""THEN 1020
1025 IFASC(A$)<48ORASC(A$)>57THEN 1015
1030 W$=W$+A$
1035 NEXT
1040 N=VAL(W$)
1045 PRINT"■NUMBER ENTERED IS ";W$
1050 PRINT"VALUE OF NUMBER IS";N
```

Line 1025 checks for a number key.

Line 1040 converts that word into a numeric variable.

If you wish to input a word of variable length the RETURN key may be used to terminate the entry as happens normally when using the INPUT statement. This will also allow you to ensure that only those characters required will be accepted.

```
10 REM ** USE RTN TO TERMINATE **
1000 W$="":PRINT"□"
1005 PRINT"■ENTER WORD ";W$
1010 POKE198,0
1015 GETA$:IFA$=""THEN 1015
1020 IFA$=CHR$(13)THENA$="":GOTO1045
1025 IFASC(A$)<65ORASC(A$)>90THEN1010
1030 W$=W$+A$
1035 PRINT"■ENTER WORD ";W$
1040 GOTO1005
1045 PRINT"■THE WORD IS ";W$
```

Line 1020 is the key line. It checks to see if the RETURN key has been pressed.

Line 1025 checks that the character is within the required range.

Of course this may be modified to accept numbers or any other characters as has been seen in the last few programmes.

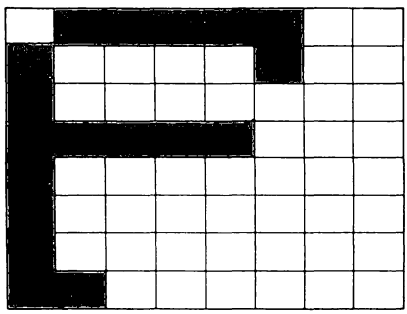
It is possible to combine this small routine to perform a very powerful set of functions. It gives scope for a wide range of applications from games to business programmes.

USER DEFINED CHARACTERS

Of the many graphics facilities available on the Commodore 64 perhaps the most useful is the capacity to define your own graphics characters.

Characters on the Commodore 64 are defined on an 8 by 8 matrix or grid:-

Eg



Each character is made up of a block of 8 bytes. For every dot in a row the corresponding bit is set to a 1 and a 0 where it is blank. The top row of the F (row 1) is a byte (01111100).

Row 1 (01111100)
Row 2 (10000100)
Row 3 (10000000)
Row 4 (11111000)
Row 5 (10001000)
Row 6 (10000000)
Row 7 (10000000)
Row 8 (11000000)

The character is stored in memory as a sequence of 8 bytes top to bottom.

The character ROM in the Commodore 64 contains the information to form all the characters available. Since this is in the Read Only

Memory you may ask how it is possible to define your own characters. This is achieved by telling the VIC chip that controls the screen display to look for the information in a different place. Because you want to store your own data this must be in RAM. As the only RAM readily available is also used by Basic it is also necessary to tell the computer that the start of your character information is the top address available. Although this restricts the availability of RAM most programmes requiring this form of graphics will still have enough room.

```
10 REM ********************************
15 REM ** SET UDC DATA TO 12288  **
20 REM ** AND PROTECT FROM BASIC **
25 REM ********************************
30 REM
40 REM
50 POKE51,255:POKE52,47
60 POKE55,255:POKE56,47
70 POKE53272,(PEEK(53272)AND240)+12
```

When you run this programme the screen will fill with garbage. This is because there is no data to form the characters in this section of memory. You can now define your own character and see what it looks like.

To define your own characters simply draw out the shape on an 8 by 8 grid. As you now know each row is effectively an 8 bit binary number and all that now has to be done is to convert that number into decimal.

This is done by adding up the relevant number of each filled square (see example below).

128	64	32	16	8	4	2	1	Decimal
	■	■	■	■	■			124
■					■			132
■								128
■	■	■	■	■				248
■								136
■								128
■								128
■	■							192

Row 1 = 64 + 32 + 16 + 8 + 4 = 124
Row 2 = 128 + 4 = 132
Row 3 = 128
Row 4 = 128 + 64 + 32 + 16 + 8 = 248
Row 5 = 128 + 8 = 136
Row 6 = 128
Row 7 = 128
Row 8 = 128 + 64 = 192

The data now has to be stored.

```
10 REM ** USER DEF 'F' **
100 POKE53272,28
110 POKE55,255:POKE56,47
120 POKE51,255:POKE52,47
130 FORN=0TO7
140 READA
150 POKE12288+N,A
160 NEXT
170 PRINT"■■■■■■■■"
180 DATA124,132,128,248,136,128,128,192
```

This programme alters the area from which the character data is read, lowers the top of memory to protect the data from being over-written and stores the data for the new character in the space normally used to hold the @ sign. As many characters as wished may be defined and used in this way. As the data starts at 12288 and is held in 8 bytes subsequent

characters will be stored in 8 byte increments.

Eg

@ 12288 to 12295
A 12296 to 12303
B 12304 to 12311

As you are no doubt aware the disadvantage of user defined characters is that they replace the normal set. One way around this is to copy the original data from the character ROM into the RAM used for your characters and only redefine those necessary.

```
10 REM ** CHAR ROM COPY **
10000 POKE51,255:POKE52,47:POKE55,255:POKE56,47
10005 REM RESERVES MEMORY FROM BASIC
10010 REM
10015 REM
10020 POKE56334,PEEK(56334) AND 254
10025 POKE1,PEEK(1) AND 251
10030 REM DISABLES KEYSCAN & ENABLE CHAR ROM
10035 REM
10040 REM
10045 FOR N=0 TO 4095
10050 POKE 12288+N,PEEK(53248+N)
10055 NEXT
10060 REM COPIES ROM CHARS INTO RAM
10065 REM
10070 REM
10075 POKE1,PEEK(1) OR 4
10080 POKE56334,PEEK(56334) OR 1
10085 REM ENABLES KEYSCAN & RESTORES CHAR ROM
```

This programme will copy the character ROM into RAM starting at 12288. However because the Commodore 64 has a 64K RAM the character ROM space is normally used for different purposes and is not immediately available. This programme disables the key scan and enables the character ROM, PEEKs the data byte by byte

and POKEs it into the RAM, restores the normal key scan functions and character ROM. Once this has been done typing POKE 53272,28 apparently has no effect. This is because the character information is now loaded.

To calculate which characters you wish to redefine look up the POKE number for the character.

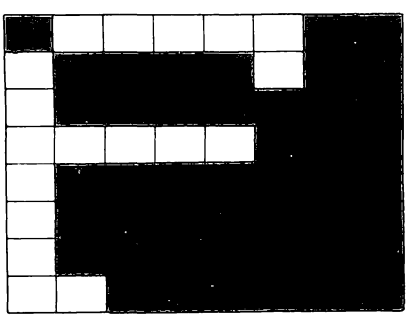
Eg

@ = 0
A = 1
Z = 26 etc

multiply this number by 8 and then add this to 12288 to find the starting location of the symbol to be replaced.

REVERSAL OF CHARACTERS

In order to create a reversed character the simplest thing to do is for every 1 put a 0 and for every 0 put a 1. Going back to the example of the F this programme uses the same data but reverses the bit pattern of each row to form a reversed character.



Row 1 = 128 + 2 + 1 = 131
Row 2 = 64 + 32 + 16 + 8 + 2 + 1 = 123
Row 3 = 64 + 32 + 16 + 8 + 4 + 2 + 1 = 127
Row 4 = 4 + 2 + 1 = 7
Row 5 = 64 + 32 + 16 + 4 + 2 + 1 = 119
Row 6 = 64 + 32 + 16 + 8 + 4 + 2 + 1 = 127
Row 7 = 64 + 32 + 16 + 8 + 4 + 2 + 1 = 127
Row 8 = 32 + 16 + 8 + 4 + 2 + 1 = 63

There is however no need to repeat these calculations to achieve a reversed character merely deduct the original calculation for each row from 255 and enter the data.

The following programme demonstrates a character reversal.

```
10 REM ** REVERSED 'F' **
100 POKE53272,28
110 POKE55,255:POKE56,47
120 POKE51,255:POKE52,47
130 FORN=0TO7
140 READA
150 POKE12288+N,255-A
160 NEXT
170 PRINT"[illegible]"
180 DATA124,132,128,248,136,128,128,192
```

Row 1 = 255 - 124 = 131
Row 2 = 255 - 132 = 123
Row 3 = 255 - 128 = 127
Row 4 = 255 - 248 = 7
Row 5 = 255 - 136 = 119
Row 6 = 255 - 128 = 127
Row 7 = 255 - 128 = 127
Row 8 = 255 - 192 = 63

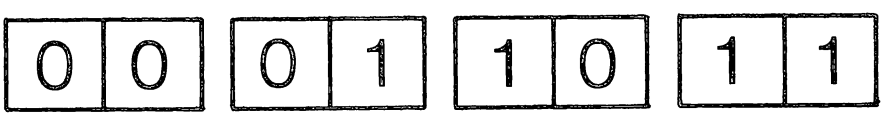
The following programme will replace the Commodore 64's normal character set with "space age" style font.

```
10 REM ** NEW CHARACTER SET
100 FOR N=0 TO 79
105 READ A
110 POKE 12288+384+N,A
115 NEXT
120 REM
125 REM READ NUMERAL DATA & STORE IN RAM
130 REM
135 REM
200 FOR N=0TO207
205 READ A:POKE12288+8+N,A
210 NEXT
215 REM READ ALPHABET DATA & STORE IN RAM
220 REM
225 REM
230 POKE53272,28
235 REM
240 REM
245 REM SWITCH TO NEW CHARACTER SET
1000 DATA126,70,74,82,102,70,126,0:REM 0
1001 DATA8,8,8,24,24,24,24,0:       REM 1
1002 DATA126,2,2,126,96,96,126,0:   REM 2
1003 DATA126,2,2,62,6,6,126,0:      REM 3
1004 DATA64,68,68,126,12,12,12,0:   REM 4
1005 DATA126,64,64,126,6,6,126,0:   REM 5
1006 DATA64,64,64,126,98,98,126,0:  REM 6
1007 DATA126,66,66,6,6,6,6,0:       REM 7
1008 DATA126,66,66,126,98,98,126,0:REM 8
1009 DATA126,70,70,126,2,2,  2 ,0:  REM 9
2000 DATA126,66,66,126,98,98,98,0:   REM A
2001 DATA124,68,68,126,98,98,126,0:  REM B
2002 DATA126,66,66,96,98,98,126,0:   REM C
2003 DATA126,66,66,98,98,98,126,0:   REM D
2004 DATA126,64,64,124,96,96,126,0:  REM E
2005 DATA126,64,64,124,96,96,96,0:   REM F
2006 DATA126,66,64,102,98,98,126,0:  REM G
2007 DATA66,66,66,126,98,98,98,0:    REM H
2008 DATA16,16,16,24,24,24,24,0:     REM I
2009 DATA2,2,2,6,70,70,126,0:        REM J
2010 DATA66,66,66,124,98,98,98,0:    REM K
2011 DATA64,64,64,96,96,96,126,0:    REM L
2012 DATA126,74,74,106,106,106,106,0:REMM
```

```
2013 DATA98,82,74,70,98,98,98,0:      REM N
2014 DATA126,66,66,70,70,70,126,0:    REM O
2015 DATA126,66,66,126,96,96,96,0:    REM P
2016 DATA126,66,66,106,106,126,8,0:   REM Q
2017 DATA124,68,68,126,98,98,98,0:    REM R
2018 DATA126,98,96,126,2,66,126,0:    REM S
2019 DATA126,16,16,24,24,24,24,0:     REM T
2020 DATA66,66,66,98,98,98,126,0:     REM U
2021 DATA66,66,66,102,36,24,24,0:     REM V
2022 DATA74,74,74,106,106,126,126,0:  REM W
2023 DATA66,66,126,24,126,66,66,0:    REM X
2024 DATA66,66,66,126,24,24,24,0:     REM Y
2025 DATA126,66,68,8,18,34,126,0:     REM Z
```

MULTI COLOUR CHARACTERS

In the preceding examples the user defined characters have been in only two colours - the background colour 0 and the character colour. This is because each bit controls one dot on the screen (pixel) as each bit can contain 0 or 1 there are only two possible combinations and hence only two colours. However the Commodore 64 will allow a different form of bit mapping where each pair of bits is treated together. This enables up to four colours as a pair of bits may have one of four values.



BKGRND 1	BKGRND 2	BKGRND 3	CHAR COLOUR
53281	53282	53283	COLOUR RAM

The colour RAM starts at 52296. Colours 0 to 7 are normal and 8 to 15 are in multi colour mode, which can be set by the PRINT command.

POKE 53270,PEEK(53270) or 16 will enable multi colour.

POKE 53270,PEEK(53270)AND239 will disable multi colour.

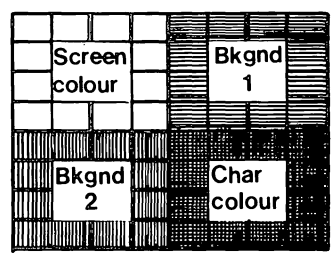
As each dot on the screen is now controlled by two bits this effectively means that the maximum horizontal resolution is cut by a factor of 2.

The first colour caused when two adjacent bits are set to 0 is the screen colour set in location 53281. The second colour is produced when the bit pattern is 01 and is held in location 53282. The third colour is produced by a 10 pattern and is stored at location 53283. The fourth colour is the character colour and is stored in the screen RAM corresponding to the position of the character. If this is between 0 and 7 the character is displayed in the normal manner (ie high resolution). If it is between 8 and 15 the character is printed multi colour the fourth colour being dependent upon the actual value. This enables both high resolution and multi colour graphics to be used on the same screen. However only colours 0 to 7 (black to yellow) may be utilised for high resolution characters.

It is important to remember when designing multi colour characters that although each bit is treated individually for calculation purposes it is the combination of two adjacent bits that determines the colour and that the pixel size is effectively doubled. Thus the character matrix is now four columns by eight rows when the multi colour mode is used.

The diagram below shows the correspondance between the bit pattern and colour displayed.

128	64	32	16	8	4	2	1	Decimal
0	0	0	0	0	1	0	1	5
0	0	0	0	0	1	0	1	5
0	0	0	0	0	1	0	1	5
0	0	0	0	0	1	0	1	5
1	0	1	0	1	1	1	1	175
1	0	1	0	1	1	1	1	175
1	0	1	0	1	1	1	1	175
1	0	1	0	1	1	1	1	175




```
10 REM ** USER DEF MULTICOLOUR **
100 PRINT"J":POKE53272,28
105 POKE55,255:POKE56,47
110 POKE51,255:POKE52,47
115 POKE53270,PEEK(53270)OR16
120 POKE53281,15:POKE53282,5
125 POKE53283,4
130 FORN=0TO7
135 READA
140 POKE12288+N,A
145 NEXT
150 FORN=0TO499:PRINT"@ ";:NEXT
155 FORN=0TO499:POKE55296+N,15:NEXT
160 FORN=500TO999:POKE55296+N,7:NEXT
165 GOTO165
1000 DATA5,5,5,5,175,175,175,175
```

In this example the colours are set at

Screen light grey (15)
background 1 green (5)
background 2 magenta (4)

The first half of the screen RAM is loaded with 15 which enables multicolour in that square and sets the fourth colour to yellow (15-8)=7

The second half is set to 7, which will only permit high-resolution characters. They appear in this half of the screen in the same form as normal user-defined characters. Thus it will be apparent why only colours 0-7 may be employed for normal user-defined characters in multi-colour mode.

SPRITES

Sprites are a very special form of user defined character and possess many unique and interesting characteristics. There are up to eight sprites normally available and each sprite may be controlled independently from all the other graphics features on the Commodore 64.

Each sprite is able to:-

Move smoothly across the screen in either direction.

Move under or over any other screen display.

Expand x 2 in either or both directions.

Move under or over another sprite.

Be defined in normal or multi colours.

Facilities also exist to detect sprite to sprite collisions and sprite to background collisions.

The steps necessary to define and use sprites may be summarised:-

1. Draw and define a sprite.
2. Locate the data in a suitable area of RAM.
3. Tell the computer where to find the data.
4. Select the attribute of the sprite.

a. colour
b. normal or multi colour
c. background priority
d. horizontal or vertical expansion

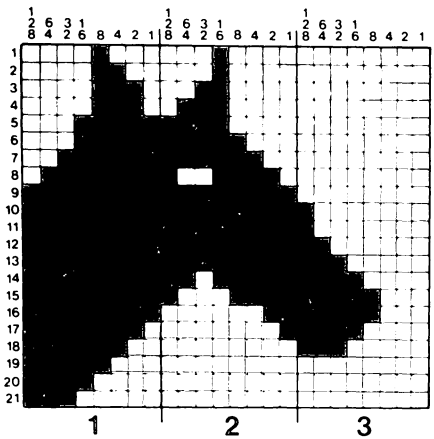
5. Enable the sprite
6. Position the sprite

Since each sprite is controlled independently from all other graphics including other sprites it is possible to have many sprites each with differing characteristics.

In order to explain in a clear manner the steps involved in sprite graphics we shall define a sprite and position it on the screen. As we move through the steps involved the techniques will be explained in detail.

STEP 1: DRAWING A SPRITE

Sprites are formed in a way analagous to user defined characters except the grid used consists of 24 horizontal and 21 vertical columns. The appearance of a sprite drawn on the grid is shown below.



Decimal		
8	16	0
12	16	0
14	48	0
14	48	0
31	240	0
31	248	0
63	252	0
127	158	0
255	255	0
255	255	128
255	255	128
255	255	192
255	255	224
255	223	240
255	143	248
255	3	248
254	1	224
252	0	192
248	0	0
240	0	0
224	0	0
1	2	3

As can be seen the horizontal rows are divided into three sets of eight squares each eight square section being treated in the normal manner for a row of user defined characters. The sprite data will therefore comprise 1 byte for each horizontal block of 8 squares and 21 bytes for each column. This makes a total of 3 x 21 or 63 bytes for each block of sprite data.

The data should be arranged Row 1 Col 1 Row 1 Col 2 Row 1 Col 3 ; Row 2 Col 1 Row 2 Col 2 Row 2 Col 3 etc reading across the sprite.

	COL 1	COL 2	Col 3
Row 1	1	2	3
Row 2	4	5	6
Row 3	7	8	9
Row 4	10	11	12
Row 5	13	14	15
.....	...	...	...
.....	...	...	...
Row 20	58	59	60
Row 21	61	62	63

TABLE: SEQUENCE TO FOLLOW WHEN DEFINING A SPRITE

The rules for determining the value foe each 8 bit row are identical to those used for user defined characters and if you are in any doubt as to how to do this read the section on user defined characters before proceeding any further with sprites.

STEP 2: STORING SPRITE DATA

Once the sprite has been drawn and the conversion of each row into a decimal number is complete the next stage is to store the information in RAM.

Before looking at how to store the data you must decide where to put it.

Each sprite will require an effective total of 64 bytes of memory. This must be within the first 16K of Ram as the VIC II chip that controls sprites can only 'see' up to location 16383. This area regrettably is also used by Basic and will limit the space for programmes to about 10K.

If you do not wish to define more than three different sprites there is a solution.

The answer is to store the data in the cassette buffer. This is a section of memory from location 828 to 1019 that is normally used to temporarily store data being transfered to and from the cassette unit. Although if the tape unit is used the sprite data will be lost it does enable you to define up to three different sets of sprite data with no 'loss' of RAM. Most people will not use the tape when running a programme anyway so there is little conflict between the two uses.

The programme below will illustrate the simplest way to store the data. The data is for the horse's head example shown above.

```
100 FORN=0TO62
105 READA
110 POKE832+N,A
115 NEXT
10000 DATA8,16,0
10005 DATA12,16,0
10010 DATA14,48,0
10015 DATA14,48,0
10020 DATA31,240,0
10025 DATA31,248,0
10030 DATA63,252,0
10035 DATA127,158,0
10040 DATA255,255,0
10045 DATA255,255,128
10050 DATA255,255,128
10055 DATA255,255,192
10060 DATA255,255,224
10065 DATA255,223,240
10070 DATA255,143,248
10075 DATA255,3,248
10080 DATA254,1,224
10085 DATA252,0,192
10090 DATA248,0,0
10095 DATA240,0,0
10100 DATA224,0,0
```

The data is held in DATA statements and read into variable A in the FOR-NEXT loop and stored in RAM starting at location 832.

The reason for locating the data there will become apparent when we examine how to tell the computer where to look for the sprite data.

If you were to use more that 3 sets of data the next most logical place would be starting at 12288. This would involve lowering the Basic pointers and would restrict the size of RAM available to Basic. Details on how to protect the data are to be found in the section on user defined characters.

STEP 3: SET SPRITE DATA POINTERS

The next step is to inform the computer where you have located the data for each sprite.

There are 8 memory locations that control the position of the sprite data one for each sprite.

2040	SPRITE 0 DATA POINTER
2041	SPRITE 1 DATA POINTER
2042	SPRITE 2 DATA POINTER
2043	SPRITE 3 DATA POINTER
2044	SPRITE 4 DATA POINTER
2045	SPRITE 5 DATA POINTER
2046	SPRITE 6 DATA POINTER
2047	SPRITE 7 DATA POINTER

The contents of each pointer register contain the actual location of the data divided by 64.

For example: data held at 832 for sprite 0 = 832 ÷ 64 = 13

The information would be set in the following manner:-

```
POKE2040,13
```

If you had the data loaded at 12288 the POKE would be:- 12288 ÷ 64 = 192 thus

POKE2040,192

sets the sprite 0 data to start at 12288.

The same set of data may be utilised by any number of sprites; all that is necessary is to set the data pointer of the sprite to the starting location desired.

For example:-
```
10 POKE2040,13
15 POKE2041,13
20 POKE2042,13
25 POKE2044,13
```

will set the first 4 sprites to the data held at 832.

You can now see why the data was loaded at 832 and not at the start of the cassette buffer. The start of the buffer is at 828 and this cannot be divided by 64 without a remainder.

STEP 4: SELECT SPRITE ATTRIBUTES

a. COLOUR

The colour of each sprite is determined by the contents of the SPRITE COLOUR REGISTER.

These are located as follows:-

53287	SPRITE 0 COLOUR
53288	SPRITE 1 COLOUR
53289	SPRITE 2 COLOUR
53290	SPRITE 3 COLOUR
53291	SPRITE 4 COLOUR
53292	SPRITE 5 COLOUR
53293	SPRITE 6 COLOUR
53294	SPRITE 7 COLOUR

Any of the 16 colours normally available may be utilised. The code for the colour required is POKEd into the relevant colour register.

Eg

POKE53287,0

and all the bits set to 1 in sprite 0 will appear in black. The bits set to 0 will be 'transparent' and will shown the colour of whatever the sprite is positioned over normally the screen colour (held in 53281).

b. NORMAL OR MULTI COLOUR

The sprite will normally be in standard mode but may also be formed in multi colour mode. The subject of multi colour sprites is fairly complex and the topic will be dealt with in depth later on.

c. BACKGROUND PRIORITY

The priority of each sprite with respect to other images on the screen may be altered by the SPRITE BACKGROUND PRIORITY REGISTER.

This is located at 53275. Each bit of this register controls one sprite and may be set to a 1 or a 0. If the bit is set to a 1 then the sprite will have a lower priority and will appear BEHIND any character on the screen. Setting the bit to a 0 will cause the sprite to be displayed IN FRONT of the other display. The priority of each sprite may be set in the following manner:-

POKE53275,(PEEK(53275)OR2↑N)
This will cause the sprite to be displayed behind other characters where N is the number (0 - 7) of the sprite.

To set the sprite to a higher priority

POKE53275,(PEEK(53275)AND(255-2↑N)

Again N is the number of the sprite.

All sprites have fixed priority over other sprites; the lower the sprite number the higher the priority. Thus sprite 0 will appear in front of ALL other sprites; sprite 5 will appear in front of sprites 6 and 7 but behind sprites 0 to 4 and sprite 7 will appear behind all other sprites.

SPRITE NUMBER	PRIORITY
0	Highest
1	
2	
3	
4	
5	
6	
7	Lowest

d. HORIZONTAL OR VERTICAL EXPANSION

Each sprite may be expanded in either the vertical or horizontal directions or even both.

This will cause the sprite to be displayed twice its normal size. The registers are again 'bitwise' where each bit controls one sprite. If the bit is set to a 1 then that sprite will expand in the direction controlled by that register.

EXPANSION REGISTER

BIT	SPRITE	SET(ie 1)
0	0	expanded
1	1	expanded
2	2	expanded
...	...	...
...	...	...
7	7	expanded

The vertical register is at 53271.
The horizontal register is at 53277.

To expand a sprite vertically:-

```
POKE53271,(PEEK(53271)OR(2↑N)
```

To return to normal:-

```
POKE53271,(PEEK(53271)AND255(2↑N)
```

where N is the number of the sprite.

To expand horizontally the formula is the same except you would POKE53277.

e. ENABLING SPRITES

Now that the characteristics of the sprites have been established you are ready to enable the sprite.

Each sprite may be enabled or turned on or disabled at will. This is controlled by the SPRITE ENABLE REGISTER which is located at 53269. In common with many registers used in the Commodore 64 each bit in the register controls one sprite.

BIT	SPRITE	CONDITION AND SPRITE STATUS
0	0	1 on 0 off
1	1	1 on 0 off
2	2	1 on 0 off
3	3	1 on 0 off
4	4	1 on 0 off
5	5	1 on 0 off
6	6	1 on 0 off
7	7	1 on 0 off

Because it is important to only affect the bit associated with the sprite you wish to use a formula is necessary to ensure that just one bit is modified:-

To enable a sprite

POKE53269,PEEK(53269)OR(2↑N)

To disable a sprite

POKE53269,PEEK(53269)AND(255-2↑N)

where N is the number (0 - 7) of the sprite in question.

You are now able to continue with the example horse in the first part of the chapter.

The complete programme up to this point will be:-

```
50 POKE53281,11
55 POKE53280,12
60 REM ** SET SCREEN & BORDER COLOUR **
100 FORN=0TO62
105 READA
110 POKE832+N,A
115 NEXT
120 REM ** STORE DATA IN TAPE BUFFER **
125 REM
```

```
130 POKE2040,13
135 REM **  SET DATA POINTER TO 832  **
140 REM
145 POKE53287,1
150 REM ** SPRITE 0 COLOUR TO WHITE  **
155 REM
160 POKE53275,PEEK(53275)OR210
165 REM ** SPRITE WILL APPEAR BEHIND **
170 REM
175 POKE53271,PEEK(53271)AND(255-210)
180 REM **   NO VERTICAL EXPANSION   **
185 REM
190 POKE53277,PEEK(53277)AND(255-210)
195 REM **  NO HORIZONTAL EXPANSION  **
200 REM
205 POKE53269,PEEK(53269)OR210
210 REM **      TURN ON SPRITE       0 **
215 REM
10000 DATA8,16,0
10005 DATA12,16,0
10010 DATA14,48,0
10015 DATA14,48,0
10020 DATA31,240,0
10025 DATA31,248,0
10030 DATA63,252,0
10035 DATA127,158,0
10040 DATA255,255,0
10045 DATA255,255,128
10050 DATA255,255,128
10055 DATA255,255,192
10060 DATA255,255,224
10065 DATA255,223,240
10070 DATA255,143,248
10075 DATA255,3,248
10080 DATA254,1,224
10085 DATA252,0,192
10090 DATA248,0,0
10095 DATA240,0,0
10100 DATA224,0,0
```

The line by line explanation is:-

50 set screen colour to grey
55 set border colour to dark grey
100 start counting loop
105 read sprite data from data statements
110 load data into RAM starting at 832
115 increment loop count
130 set data pointer to 832 (832 ÷ 64 = 13)
for sprite 0
145 set sprite 0 colour to white
160 ensure sprite 0 priority is low
175 ensure sprite will appear normal size
190 ensure sprite will appear normal size
205 enable sprite 0
10000 to 10100 contain the sprite data arranged in columns and rows (compare with the diagram of the horse's head sprite).

With all this done if the programme is RUN all that will happen is that the screen and the border will change colour. This is because the sprite has not yet been positioned on the screen.

STEP 5: POSITIONING SPRITES ON THE SCREEN

For the purposes of sprite positioning the screen may be viewed as consisting of 255 vertical and 344 horizontal positions each sprite being located by specifying the X (horizontal) and Y (vertical) coordinates.

For the purposes of coordinate calculation the sprite is taken as starting from the top left hand corner. This is regardless of whether the sprite has been expanded or not.

With a sprite at its normal size, positions 0-29 and 250-255 place the sprite outside the viewing area, and are thus the extremes to which a sprite may be moved, and still remain visible.

In the horizontal direction the limit is set at 344 when the sprite just disappears off the right hand side of the screen.

To position the sprite so that all of it is just visible the coordinates are 50 - 229 vertically and 24 - 320 horizontally.

The value corresponding to the position of the sprite is held in the sprite position registers:-

LOCATION	SPRITE AND DIRECTION	
53248	0	Horizontal X
53249	0	Vertical Y
53250	1	Horizontal X
53251	1	Vertical Y
53252	2	Horizontal X
53253	2	Vertical Y
53254	3	Horizontal X
53255	3	Vertical Y
53256	4	Horizontal X
53257	4	Vertical Y
53258	5	Horizontal X
53259	5	Vertical Y
53260	6	Horizontal X
53261	6	Vertical Y
53262	7	Horizontal X
53263	7	Vertical Y

In addition there is the X direction MSB register located at 53264. This will be looked at shortly.

To position a sprite you need to POKE the value corresponding to the desired position into the position registers for that sprite. In the previous example sprite 0 was defined and that sprite will now be positioned on the screen.

POKE53248,100:POKE53249,100

will cause the horse's head to appear on the screen.

It is now easy to cause the sprite to move from the top to the bottom of the screen thus demonstrating the ability to animate displays. Add the lines below to the programme and RUN it to observe the effect.

```
200 REM
205 POKE53269,PEEK(53269)OR2^0
210 REM **      TURN ON SPRITE       0 **
215 REM
220 POKE53248,100
225 FORN=50 TO229
230 POKE53249,N
235 FORT=1TO50:NEXT
240 NEXT
245 REM * MOVE HORSE FROM TOP TO BOTTOM *
250 REM
```

220 sets the horizontal position
225 starts the counting loop
230 POKEs the vertical position into the Y register
235 slows down the movement
240 increments the loop

To move the sprite back to the top of the screen simply alter line 225

225 FORN=229TO50STEP-1

The horse will now move from the bottom to the top of the screen.
By altering the programme so that the vertical position is fixed and changing the horizontal position in a FOR-NEXT loop it is possible to move the sprite across the screen.

```
200 REM
205 POKE53269,PEEK(53269)OR210
210 REM **      TURN ON SPRITE       0 **
215 REM
220 POKE53249,100
225 FORN=24 TO255
230 POKE53248,N
235 FORT=1TO50:NEXT
240 NEXT
245 REM * MOVE HORSE FROM LEFT TO RIGHT *
250 REM
```

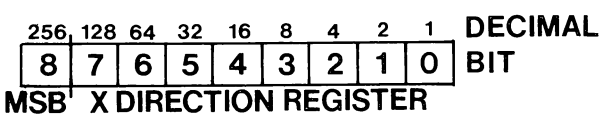
You will now see the horse move from left to right. However the motion stops about three-quarters of the way across. You will recall that the screen is 24 to 320 'steps' across. Because the position register is one byte and one byte on the Commodore 64 is 8 bits wide the maximum decimal value that it can contain is 255. If you try to POKE a number greater than 255 into the register you will be rewarded with an error message:-

?ILLEGAL QUANTITY ERROR IN 230

Fortunately the designers knew of this limitation and provided a way around it.

X MSB REGISTER

The X direction MSB or Most Significant Bit register is located at 53264 and contains the most significant bit of a nine bit byte. In effect when you POKE a number into any of the X direction registers you are setting the 8 lowest bits of a nine bit byte:

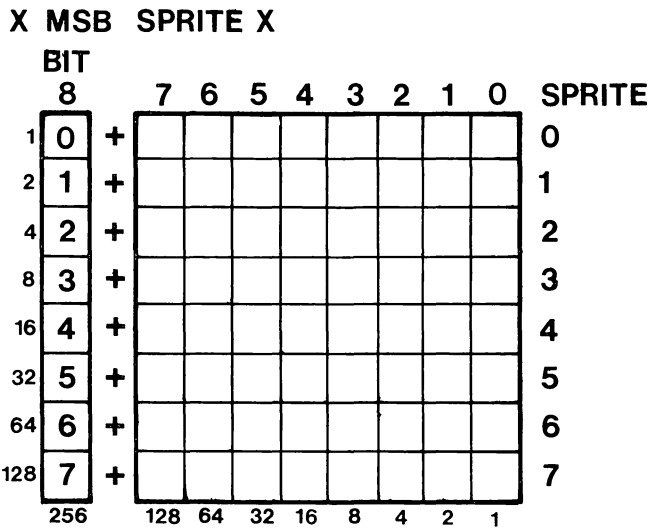


Since the maximum number that can be held in a 9 bit number is

256 + 128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = 511

you can position your sprites wherever you wish.

Each bit in the MSB register is effectively the highest or 8th bit for each of the 8 sprite X direction registers and may be viewed thus:-



Whenever it is desired to position a sprite beyond 255 you must set the MSB bit corresponding to that sprite and clear the normal X register to 0 and load in the new value.

The existing register contents must be cleared because when the MSB is set to 1, 256 is added to the contents of the normal register which would place the sprite off the screen.

For example:-

MSB	NORMAL REGISTER	COMBINED VALUE
0	254	254
0	255	255
1	255	511 * not a good idea
1	0	256
1	1	257
1	64	320 *maximum value

The relationship between the total value and the condition of the MSB can be clearly seen in the table above.

HOW TO SET THE MSB

Clearly you must ensure that only the MSB for the sprite in question is altered or all the other sprites will move as well. The formula below will enable you to do this.

To set MSB to 1 and clear X register
POKE53264,PEEK(53264)OR2 N:POKE|X REG|,0

where X REG is the appropriate X position register and N is the number of the sprite.

To clear the MSB and set the normal register:-

POKE53264,PEEK(53264)AND(2 N):POKE|X REG|,255

Again X REG is the normal X register location and N is the number of the sprite.

You can now modify the programme to move the horse all the way across the screen.

```
255 POKE53264,PEEK(53264)OR2↑0
260 REM ** SET MSB FOR SPRITE 0 **
265 REM
270 POKE53248,0
275 REM ** CLEAR SPRITE 0 X REGISTER **
280 REM
285 FORN=1TO64
290 POKE53248,N
295 FORT=1TO50:NEXT
300 NEXT
305 REM **   MOVE FROM 255 TO 320     **
```

255 set MSB for sprite 0
270 clear normal X register
280 start loop
285 load register with position
290 slow down movement
295 increment loop

By combining the directions of movement the sprite can be made to move in any direction including diagonally or even in circles.

```
220 PRINT"□":FORV=0TO(2*π)STEP.05
225 X=40*SIN(V):X=X+150
230 Y=40*COS(V):Y=Y+150
235 POKE53248,X
240 POKE53249,Y
245 NEXT
```

220 clears the screen and starts counting loop
225 defines X coordinate of circle
230 defines Y coordinate of circle
235 POKEs X coordinate into register
240 POKEs Y coordinate into register
245 increments loop

ADDITIONAL FEATURES

If you wish to expand the sprite all that is required is a modification to line 175:-

175 POKE53271,PEEK(53271)OR2↑0

and the sprite will be double height.

Changing line 190 will double the width:-

190 POKE53277,PEEK(53277)OR2↑0

```
50 POKE53281,11
55 POKE53280,12
60 REM ** SET SCREEN & BORDER COLOUR **
100 FORN=0TO62
105 READA
110 POKE832+N,A
115 NEXT
120 REM ** STORE DATA IN TAPE BUFFER **
125 REM
130 POKE2040,13
135 REM **  SET DATA PONTER TO 832  **
140 REM
145 POKE53287,1
150 REM ** SPRITE 0 COLOUR TO WHITE  **
155 REM
160 POKE53275,PEEK(53275)OR2↑0
165 REM ** SPRITE WILL APPEAR BEHIND **
170 REM
175 POKE53271,PEEK(53271)OR2↑0
180 REM **  SET VERTICAL EXPANSION   **
185 REM
190 POKE53277,PEEK(53277)OR2↑0
195 REM ** SET HORIZONTAL EXPANSION  **
200 REM
205 POKE53269,PEEK(53269)OR2↑0
210 REM **      TURN ON SPRITE       0 **
215 REM
220 POKE53248,150
225 PRINT"■"
230 FORN=0TO6:PRINT:NEXT
235 FORN=0TO199:PRINT"■■";:NEXT
240 FORN=0TO255
245 POKE53249,N
250 FORT=1TO50:NEXT
255 NEXT
260 END
```

If the above modifications are added to the horse's head programme in addition to the expansion of the sprite in both directions you will see the sprite 'disappear' behind a barrier halfway down the screen.

If line 160 is changed to:-

160 POKE53271,PEEK(53271)OR(255-2 N)

the sprite will move over the barrier.

SIMPLE ANIMATION

The most straightforward way to animate a sprite is to store two or more sets of data for each position of the sprite and change its shape by altering the pointer to select different data. This is illustrated in the programme below:-

```
10 REM ********************************
15 REM **  SPRITE ANIMATION EXAMPLE  **
20 REM ********************************
25 REM
30 REM
100 FORN=0TO62:READA:POKE832+N,A:NEXT
105 FORN=0TO62:READA:POKE896+N,A:NEXT
110 POKE53269,PEEK(53269)OR210
115 POKE53248,100:POKE53249,100
120 POKE2040,13
125 FORT=1TO500:NEXT
130 POKE2040,14
135 FORT=1TO500:NEXT
140 GOTO120
10000 DATA0,24,0
10001 DATA0,60,0
10002 DATA0,126,0
10003 DATA0,255,12
10004 DATA0,179,12
10005 DATA0,255,12
10006 DATA0,60,12
10007 DATA0,24,12
```

```
10008 DATA127,251,252
10009 DATA127,247,252
10010 DATA96,239,0
10011 DATA96,223,0
10012 DATA120,60,0
10013 DATA120,60,0
10014 DATA127,204,0
10015 DATA31,204,0
10016 DATA0,12,0
10017 DATA0,12,0
10018 DATA0,12,0
10019 DATA0,15,0
10020 DATA0,15,0
10100 DATA0,24,0
10101 DATA0,60,0
10102 DATA0,126,0
10103 DATA48,255,0
10104 DATA48,219,0
10105 DATA48,255,0
10106 DATA48,60,0
10107 DATA48,24,0
10108 DATA63,233,252
10109 DATA63,239,252
10110 DATA0,247,12
10111 DATA0,251,12
10112 DATA0,60,12
10113 DATA0,60,12
10114 DATA0,60,12
10115 DATA0,60,0
10116 DATA0,60,0
10117 DATA0,60,0
10118 DATA0,60,0
10119 DATA0,255,0
10120 DATA0,255,0
```

10000 - 10020 hold the data for the 1st shape
10100 - 10120 hold the data for the 2nd shape
100 reads and stores the data for the 1st shape
105 reads and stores the data for the 2nd shape
110 turns on sprite 0
115 positions the sprite on the screen
120 sets the data pointer to the 1st shape

125 delays the animation
130 sets the data pointer to the 2nd shape
135 delays the animation
140 returns to continue the effect

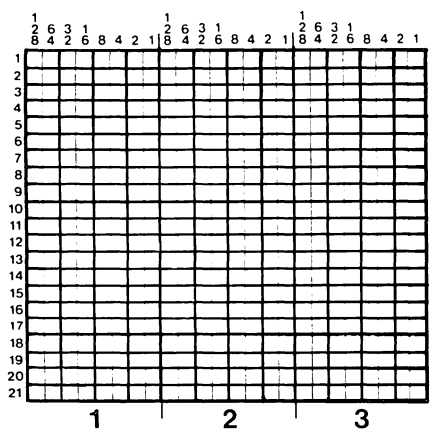
In the examples above the data statements have all been limited to 3 sets per line. This is to show more clearly the relationship between the columns and rows on the drawn sprite and their location in memory.

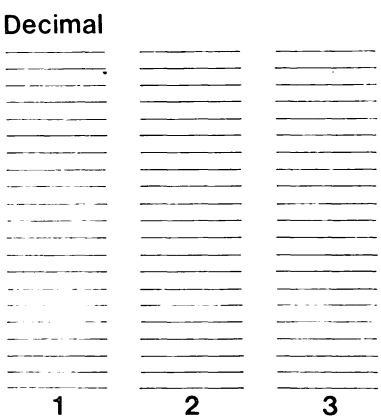
It is strongly recommended that until you are quite familiar with the relationship this method is used.

Once the programme has been written and the sprite data is known to be correct the data statements should be 'run together' to conserve memory space.

MULTI COLOUR SPRITES

All the sprites dealt with so far have been in the standard or high resolution mode. The Commodore 64 also permits sprites to be defined in multi colour mode. This will allow up to four colours to be used in each sprite. The sacrifice as usual is that as two bits are required to select the colour the horizontal resolution is halved.





As will be seen in the above diagram the sprite now consists of 12 horizontal and 21 vertical sections. The bit pattern in each section controls the colour of that area. The relationship is:-

BIT PATTERN	COLOUR DISPLAYED
00	Transparent (screen)
01	Sprite multi colour 1 (53285)
10	Sprite normal colour
11	Sprite multi colour 2 (53286)

It will be apparent that three of the colours are common to each sprite and one colour may be selected independently.

The SPRITE MULTI COLOUR ENABLE REGISTER controls each sprite and any all or no sprites may be displayed in multi colour mode.

The register is located at 53276 and once again each bit controls one sprite, bit 0 controlling sprite 0 and bit 1 controlling sprite 1 etc.

To set a sprite to multi colour the corresponding bit in the register must be set to a 1 or to display a sprite in normal mode set to a 0.

Because it is essential to only affect the bit responsible for the sprite in question the familiar formula must be employed:-

```
POKE53276,PEEK(53276)OR2↑N
```

where N is the number of the sprite to turn a sprite into multi colour mode.

```
POKE53276,PEEK(53276)AND255-(2↑N)
```
will restore a multi colour sprite to normal mode.

The colours are selected by the contents of four registers.

Bits 10 will be set in the normal sprite colour register thus permitting each sprite to have a different colour with this bit combination. The register used will depend on the number of the sprite and is the same as that used in normal mode.

Bits 11 will be set to SPRITE MULTI COLOUR 2 held in 53286. The codes are the same for all sprites and may contain any colour code from 0 to 15.

The programme below will demonstrate what a multi colour sprite may look like.

```
10 REM ************************************
15 REM ** MULTICOLOUR SPRITE EXAMPLE **
20 REM ************************************
25 REM
30 REM
100 FORN=0TO62:READA:POKE832+N,A:NEXT
105 POKE2040,13
110 POKE53276,PEEK(53276)OR2↑0
115 POKE53285,8: REM * M-COLOUR 1 01 **
120 POKE53286,6: REM * M-COLOUR 2 11 **
125 POKE53281,15:REM * SCREEN COL 00 **
130 POKE53287,7: REM * SPRITE COL 10 **
135 POKE53269,PEEK(53269)OR2↑0
140 POKE53271,PEEK(53271)OR2↑0
145 POKE53277,PEEK(53277)OR2↑0
150 POKE53248,140
155 POKE53249,120
10000 DATA0,20,0
10001 DATA0,20,0
10002 DATA0,85,0
10003 DATA0,105,4
10004 DATA0,105,4
10005 DATA0,85,12
10006 DATA0,20,12
10007 DATA0,255,12
10008 DATA63,255,252
10009 DATA63,235,252
```

```
10010 DATA48,235,0
10011 DATA48,255,0
10012 DATA48,60,0
10013 DATA48,60,0
10014 DATA16,60,0
10015 DATA16,40,0
10016 DATA0,40,0
10017 DATA0,40,0
10018 DATA0,40,0
10019 DATA0,105,0
10020 DATA0,105,0
```

COLLISION DETECTION

The facilities exist for the computer to detect whenever a sprite collides with another sprite or any other display on the screen.

The collision status for each sprite is held as a bit in two registers the SPRITE COLLISION REGISTERS.

One register holds sprite to sprite collision status the other sprite to background status.

They are located at:-

53278 SPRITE-SPRITE REGISTER
53279 SPRITE-BACKGROUND REGISTER

The bit corresponding to the similarly numbered sprite is set to a 1 if that sprite is involved in a collision.

To use the registers the formula below is used:-

```
IFPEEK(53278)AND(2↑N)=(2↑N)THEN XXXXXX
```

This checks if sprite N has had a collision: if it has then do XXXXXX where XXXXXX is whatever action is required.

By changing 53278 to 53279 it is possible to check sprite background impacts.

NOTES ON COLLISIONS

The collision status will be set even if the sprite is off the screen at the time of impact.

The register is set when the collision occurs and remains set until it is PEEKed. Once it has been PEEKed IT IS CLEARED.

When a collision occurs between 2 sprites the bits for each sprite are set.

In multi colour mode the bit pattern 01 (multi colour 1) although visible is considered to be transparent as far as collisions are concerned so you must ensure that the programme will not attempt to detect a collision with this pattern.

THE SCREEN

The screen layout of the Commodore 64 is normally arranged into 40 columns of 25 rows. The starting location of the screen is 1024 and it continues to 2023.

When you POKE a number to the screen you tell the operating system what character to put on the screen at that position. The actual shape of the character is stored in the character ROM or if you are using user defined characters the appropriate place in RAM. The chapter on user defined characters gives more information on how to form your own characters and where they should be placed in memory.

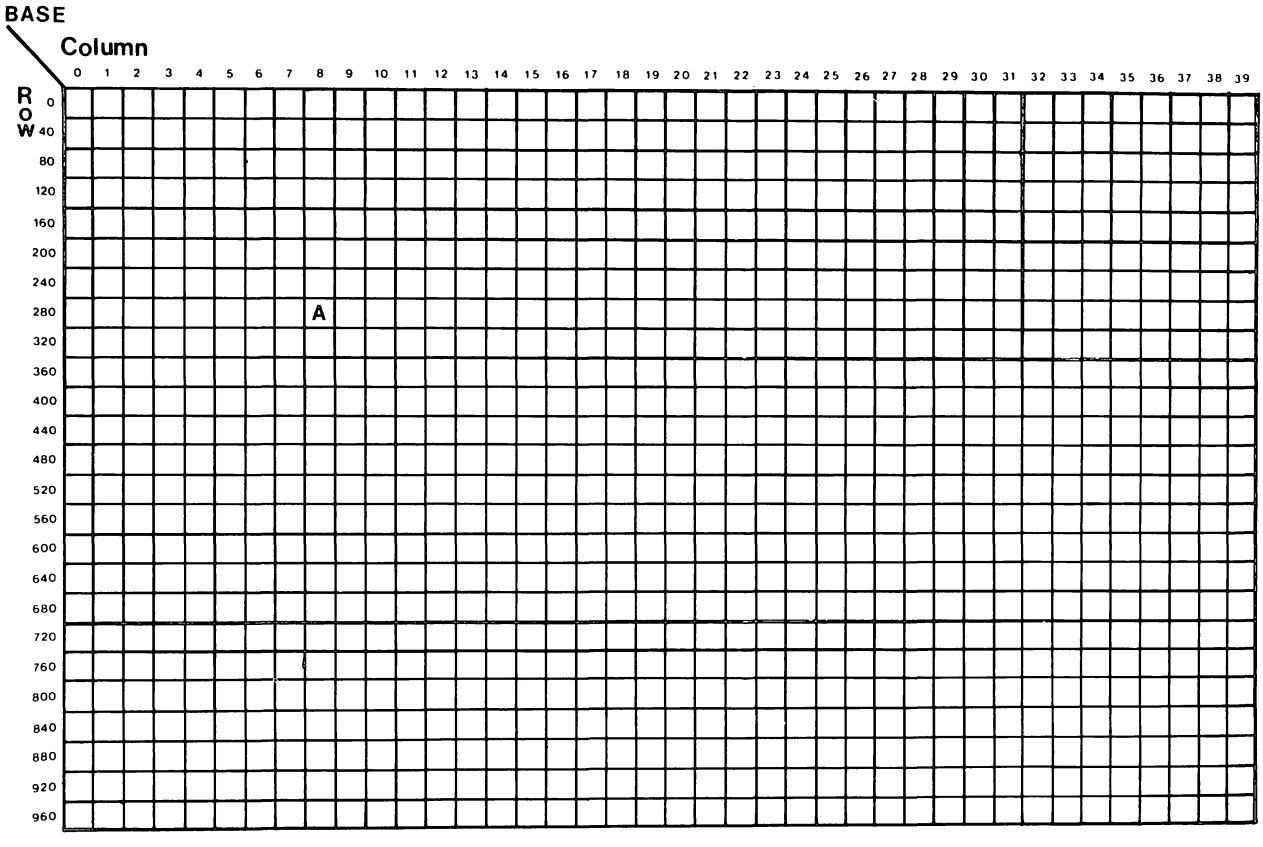
The figure below illustrates the correspondence between the position on the screen and the memory location POKEd.

By adding the column and row values obtained from the following chart and adding them to the starting address of the RAM concerned ie 1024 for characters or 55296 for colour it is easy to position your character and colour anywhere on the screen. Eg to put a letter 'A' in red at the position marked:-

```
Row      = 280
Column =   8 = 288
+ Screen 1024 + 288
+ Colour 55296 +288

POKE 1024 + 288,1      (A)
POKE 55296 + 288,2     (red)
```

The colour information is also stored in RAM and starts at 55296. In this area the number of the colour to be displayed is held. By POKEing the corresponding location in the colour RAM you can change the colour of whatever character is displayed.



COLOUR BASE 55296 CHAR BASE 1024

Because two pieces of information are required for every character displayed on the screen (character and colour) whenever you wish to modify the screen display by POKEing you should always make sure that both the colour and the character shape are as you wish them to be. This might sound obvious but a firm understanding of the way in which the screen is handled will allow you to get to grips with the more complicated aspects of the changes that are possible to the way in which the screen is used.

Consider for a moment that you have formed a complicated display either by POKEing the screen RAMs or by PRINTing the information. The chances are that you will want to use this display in various parts of your programme. Instead of having to repeat the original process because the screen information is stored in RAM regardless of how the display was formed it is possible to 'move' the information to another part of the computer's memory and to bring the display back whenever you wish to use it again. It is also possible to store the screen on disk or cassette.

The programmes below will move the entire contents of both the screen and colour RAMs to a 'dead' area of the Commodore 64's memory. This area is not used by the computer and will not affect the spare RAM for Basic.

```
10 REM ** SCREEN MOVE PROGRAM **
20 GOSUB1000:REM SAVE SCREEN
25 PRINT"SCREEN INFORMATION MOVED"
30 END
1000 FORN=0TO999
1005 POKE49152+N,PEEK(1024+N)
1010 POKE50176+N,PEEK(55296+N)
1015 NEXT
1020 RETURN
```

The 4K block starts at 49152 and continues to 53247. This will actually hold two complete screens as each screen requires 1000 bytes for characters and 1000 bytes for colour information.

This programme will restore the 'saved' screen.

```
10 REM ** SCREEN RESTORE PROGRAM **
20 GOSUB1000:REM RESTORE SCREEN
25 PRINT"SCREEN RESTORED"
30 END
1000 FORN=0TO999
1005 POKE1024+N,PEEK(49152+N)
1010 POKE55296+N,PEEK(50176+N)
1015 NEXT
1020 RETURN
```

The screen information is copied into the 1000 bytes starting at 49152 and the colour information to the 1000 bytes starting at 50176. If you wish to store a second screen the locations would be 51200 for the characters and 52224 for the colour data.

The main drawback of these routines is that they are slow to transfer the data. This is fine if you like watching the screen build up byte by byte but is of little use in fast games for example.
The way around this is to work in machine code.

Although explaining machine code is outside the scope of this book the examples shown here will perform the same functions as the above programmes only a lot faster.

Quite simply the FOR-NEXT loop reads from the DATA statements the machine code programme and POKEs it into memory. When you wish to save the screen simply GOSUB10000 and to restore it GOSUB11000. The POKEs in these lines determine whether you are saving or restoring the screen.

```
50 POKE51,175:POKE52,159
55 POKE55,175:POKE56,159
60 IFPEEK(40880)<>8THENGOSUB50000
10000 POKE40896,192:POKE40900,4
10005 POKE40904,196:POKE40908,216
10010 SYS40880:RETURN
10015 REM
10020 REM ** SAVES SCREEN **
10025 REM
11000 POKE40896,4:POKE40900,192
11005 POKE40904,216:POKE40908,196
11010 SYS40880:RETURN
11015 REM
11020 REM ** RESTORES SCREEN **
11025 REM
50000 FORN=0TO79:READA:POKE40880+N,A:NEXT
50005 RETURN
50010 REM
50020 REM ** LOAD MACHINE CODE **
50025 REM
60000 DATA8,169,0,133,180,133,247,133,167
60005 DATA133,169,133,171,133,189,169,4
60010 DATA133,181,169,192,133,248,169,216
60015 DATA133,168,169,196,133,170,160,0
60020 DATA177,247,145,180,177,169,145,167
60025 DATA200,196,189,240,3,76,209,159,230
60030 DATA171,230,181,230,248,230,168,230
60035 DATA170,165,171,201,3,240,4,201,4,240
60040 DATA7,169,233,133,189,76,207,159,40
60045 DATA96,79,78,148
```

In both cases the information is transfered to 49152 and 50176. If you wish to store two screens the additions to the machine code routine shown below will enable you to do so. Although it still takes time to POKE the machine code in this can be done once and then forgotten about.

The first line will ensure that Basic will not overwrite the code with variables. The amount of memory lost is about 80 bytes a, very small

price to pay for the increase in speed. One final point on this machine code routine it will work with the screen data moved by the previous Basic routine as well because it locates the information in the same place.

```
12000 POKE40896,200:POKE40900,4
12005 POKE40904,204:POKE40908,216
12010 SYS40880:RETURN
12015 REM
12020 REM ** SAVES SCREEN TWO **
12025 REM
13000 POKE40896,4:POKE40900,200
13005 POKE40904,216:POKE40908,204
13010 SYS40880:RETURN
13015 REM
13020 REM ** RESTORES SCREEN TWO **
```

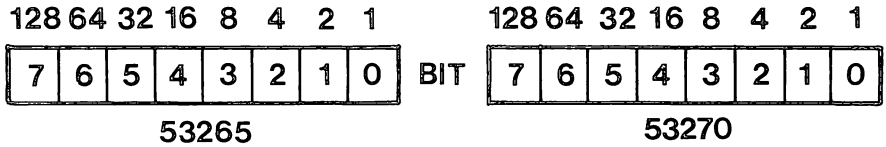
SCREEN SCROLLING

The Commodore 64 will also support "scrolling". This enables the text or graphics on the display to be moved smoothly horizontally or vertically.

Unfortunately the hardware that permits scrolling is only capable of moving the display by 8 pixels which is one character. This means that the bulk of the movement must be done with a machine code programme.

The VIC II chip which is responsible for the generation of the display will permit the screen to be in any one of eight horizontal or vertical positions. It is this feature that allows scrolling. In order to provide an area for the characters to come from the first step is to select a 38 column by 24 row display. The two control registers in the VIC II chip which control the display format are located at 53270 (columns) and 53265 (rows). In common with many registers in the Commodore 64 the registers

are also used for other purposes and each bit controls a different aspect of the display control. The bit concerned is bit 3 and it must be ensured that only this bit is affected.



VIC II control registers

53265	53270
Bit 0)	Bit 0)
1) Y scroll pos	1) X scroll pos
2)	2)
3 24 or 25 row display 1 = 25	3 38 or 40 columns 1 = 40 cols
4 Blank screen to border 1 = on	4 Multi colour mode 1 = on
5 Bit map flag 1 = on	5 Must be set to 0 VIC reset
6 Extended colour flag 1 = on	6 Unused
7 Bit 8 of raster compare	7 Unused

To go into 38 column mode use:-

```
POKE53270,(PEEK(53270)AND247)
```

To restore to 40 columns use:-

```
POKE53270,(PEEK(53270)OR8)
```

To select 24 rows use:-

```
POKE53265,(PEEK(53265)AND247)
```

To return to 25 rows use:-

```
POKE53265,(PEEK(53265)OR8)
```

The screen does not really become smaller the border expands to cover part of the screen. The characters not visible are merely hidden behind the mask.

The position of the screen and hence the illusion of scrolling are controlled by the first 3 bits in the control registers 53270 and 53265.

In order to move the display you need to POKE into the registers a number between 0 and 7. This corresponds to 0 (far left or top) 7 (far right or bottom).

```
10 REM ** X SCROLL EXAMPLE **
20 POKE53270,PEEK(53270)AND247
30 FORN=0TO7
40 POKE53270,(PEEK(53270)AND248)+N
50 FORT=1TO500:NEXT
60 NEXT
```

Line 20 sets 38 column mode.

Line 30 starts the FOR-NEXT loop that controls the scroll position.

Line 40 POKEs the control register with the position data.

Line 50 is simply a time delay loop so that you can see the motion more clearly.

Line 60 causes the next loop to occur.

You will see when the programme is run that the screen shrinks and all the characters on the screen move slowly from left to right. Because the hardware only allows a smooth scroll of 8, pixels to move the display any further to the right requires a little programming. However as this must appear to be instantaneous machine code is the only suitable method.

```
10 REM ** MACHINE CODE SCROLL **
100 FORN=0TO25
105 READA
110 POKE49152+N,A
115 NEXT
120 REM ** MACHINE CODE LOADED AT 49152 **
125 PRINT"□---------- X SCROLL EXAMPLE ----------
130 POKE53270,PEEK(53270)AND247
135 FORN=0TO39
140 FORX=0TO7
145 POKE53270,(PEEK(53270)AND248)+X
150 FORT=1TO300:NEXT
155 NEXT:SYS49152
160 NEXT
1000 DATA8,173,39,4,133,245,162,38,189
1005 DATA0,4,157,1,4,202,224,255,208,245
1010 DATA165,245,141,0,4,40,96
```

This programme loads the small machine code routine by READing the data from the DATA statements and POKEing it into the relevant area of RAM.

The machine code shifts all the characters on the top line of the screen one position to the right after the Basic has moved the display as far as it can. The slight flicker when this occurs is due to Basic not being fast enough to reset the position to 0. If this whole programme was in machine code the motion would be very smooth.

So far you have seen the horizontal scroll from left to right. It is very easy to scroll in other directions but in order to move all the way across the screen some machine code is inevitable.

Scrolling can thus be summarised:-

1. Set the screen format (38 or 40 cols 24 or 25 rows).

2. Load the control register(s) with the starting position.

3. Place the first characters to be moved in the masked portion of the screen.

4. Alter the position by suitable incrementing or decrementing the register until it is in the final position.

5. SYS away to the machine code routine to move the screen accordingly.

6. If further scrolling is required go back to step 2 and continue the process.

EXTENDED COLOUR MODE

Another of the modes that the screen can operate under is the EXTENDED COLOUR mode. Under normal circumstances the character is displayed in the character colour POKEd or PRINTed against the background colour of the screen. In extended colour mode however you may select the colour that the character is displayed on.

For example you may wish to have a yellow character displayed against a white background on a green screen.

The mode is controlled yet again by one bit in the VIC II's control register at location 53265. Needless to say only the bit required must be affected. This is done as follows:-

To enable extended mode:-

```
POKE53265,(PEEK(53265)OR64)
```

To disable extended colour mode:-

```
POKE53265,(PEEK(53265)AND191)
```

The background colour is selected by a combination of the character code and the contents of four colour registers. This means that you are limited to displaying only the first 64 characters because bits 7 and 6 of the character code actually select the colour of the background.

CHARACTER CODE	BIT 7	BIT 6	COLOUR REGISTER
0 to 63	0	0	0 53281
64 to 127	0	1	1 53282
128 to 191	1	0	2 53283
192 to 255	1	1	3 53284

The values held in the colour registers control the background colour of the character and are the same as the values used when POKEing colour to the screen.

The following example will display all the characters and illustrate the way in which extended colour graphics are implemented.

```
10 REM ** EXTENDED COLOUR MODE **
100 PRINT"□":POKE53280,1:POKE53281,1
105 FORN=0TO255
110 POKE1024+N,N
115 POKE55296+N,0
120 NEXT
125 REM ** CLEAR SCREEN SET SCREEN TO WHITE
130 REM ** DISPLAY ALL CHARACTERS IN BLACK
140 PRINT"■■■■■■■■■■■■■■■■STANDARD MODE"
150 FORT=1TO5000:NEXT:REM ** WAIT A WHILE
155 PRINT"■■■■■■■■■■■■■■■■EXTENDED COLOUR MODE"
160 POKE53282,4
165 POKE53283,5
170 POKE53284,7
175 REM ** SET BACKGROUND COLOUR
180 POKE53265,(PEEK(53265)OR64)
185 REM ** ENTER EXTENDED COLOUR MODE
```

When you run the programme the computer will display all the available upper case characters in black on a white screen. After about five seconds the display will change to extended colour mode.

The first 64 characters will remain unchanged as the register 0 (see table above) is also the screen colour register. The next 64 characters will however be displayed against background colour 1 held in 53282. It should be noted that the character codes stored in the screen memory have not changed, the VIC chip is merely interpreting the information in a different manner. The next 64 characters are displayed against background colour 2 as held in location 53283 and the last 64 characters will be displayed against background colour 3 in location 53284.

If you change the contents of any of these registers all the characters using the register will change their background colour at once. If you display the characters in the same colour as the background colour they will appear to be invisible, however change the background colour and they will all immediately jump back into view.

HIGH RESOLUTION BIT MAPPING

By now you will be aware that the screen of the Commodore 64 consists of 40 rows of 8 pixels horizontally and 25 rows of 8 pixels vertically. This means that the maximum resolution possible is 8 x 40 or 320 dots across the screen and 25 x 8 or 200 dots down.

In many applications it is far easier to be able to specify single dot locations on the screen by means of two coordinates called X (horizontal) and Y (vertical). This is possible on the

Commodore 64 by entering the BIT MAP mode and a little programming.

Bit mapping is so called because there is a section of memory where each bit is directly mapped onto the screen. This is different from the normal mode whereby the screen RAM only holds a pointer to the shape of the character to be displayed in that position.

The main disadvantage of bit mapping is that in order to map the entire screen 8K of RAM is required to store the information. This is because there are 40 rows of 8 pixels which means each row will require 320/8 or 40 bytes and as there are 25 x 8 or 200 rows of pixels down resulting in 200 x 40 or 8000 bytes to map the complete screen.
Before getting involved in the details of how to map the screen the decision must be made on where in memory to locate the bit map of the screen. As you will want to keep as much RAM free for Basic as possible and the map will require 8000 bytes the most satisfactory position is to start the screen at 8192. This will leave about 6K for the Basic programme.

Although this may not sound very much most of the techniques used are simple, repetitive routines that do not occupy much space and in any event machine code will have to be used if graphics are to be displayed at any speed. Machine code programmes may be stored almost anywhere in RAM and would normally be placed in the space above the section used for the map.

To position the map at 8192 the VIC chip must be told where to obtain the data. This is done with a POKE.

```
POKE53272,(PEEK(53272)OR8)
```

To prevent the map being overwritten by Basic it is necessary to alter the memory pointers.

EG

```
POKE51,255:POKE55,255:POKE52,31:POKE56,32
```

This will force Basic to end at 8191 and thus it will not interfere with the screen area.

Now that you have an area of RAM for the high resolution screen the next step is to enter the bit map mode with:-

```
POKE53265,(PEEK(53265)OR32)
```

If you enter the programme below and RUN it you will see the screen fill with absolute rubbish. This is because the bit map area is full of random information. To clear the screen ready to work on you will need to eliminate the garbage. The line below will do this.

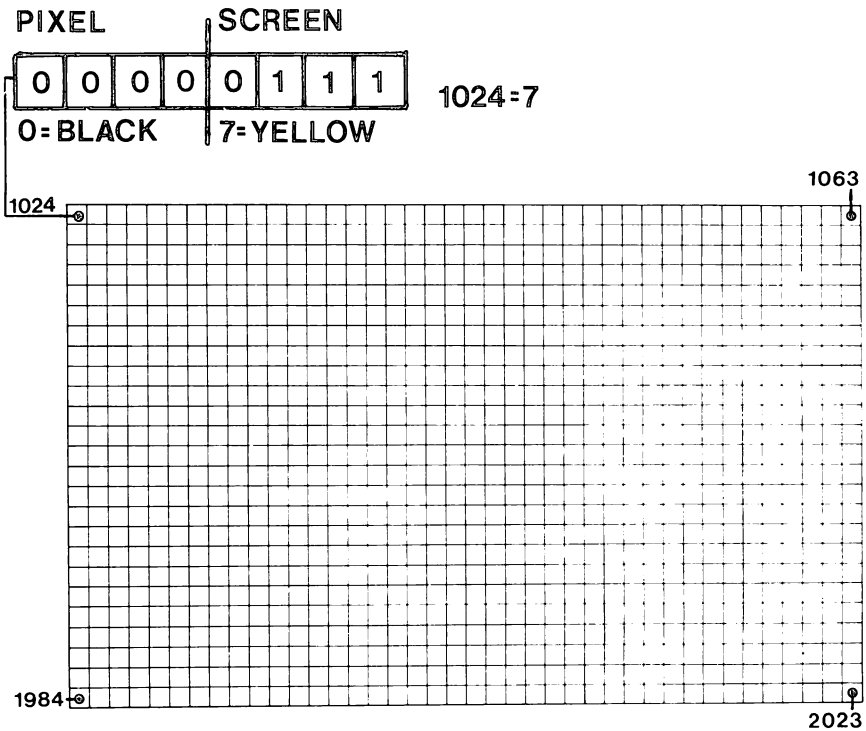
```
10 REM ** ENTER BIT MAP **
15 POKE53272,(PEEK(53272)OR8)
20 POKE51,255:POKE55,255:POKE52,31:POKE56,31
25 POKE53265,PEEK(53265)OR32


30 FORN=8192TO8192+8000:POKEN,0:NEXT
```

The screen has now cleared to black but there are still blocks of colour that look like medal ribbons. This is because the area that was used to hold the character information 1024 to 2023 is used to hold the colour data in bit map mode.

The relationship between the colours displayed and the contents of the memory location are shown below.

The first four bits numbered 0 to 3 hold the colour of bits set to 0 in the map area and the last four bits numbered 4 to 7 hold the colour of the bits set to 1.



For example to set a yellow screen with black 'writing'

Yellow is code 7 (binary 00000111)

Black is code 0 (binary 00000000)

The last bits will be 0000 (black) value = 16 x 0 = 0

The first bits will be 0111 (yellow) value = 7

Put together it looks like this: 00000111 which

is 1 + 2 + 4 which is 7 decimal. This diagram will help you to set the colour to the required values.

PIXEL(SET TO 1)		COLOUR(SCREEN SET TO 0)	
0 x 16 =	0	BLACK	0
1 x 16 =	16	WHITE	1
2 x 16 =	32	RED	2
3 x 16 =	48	CYAN	3
4 x 16 =	64	PURPLE	4
5 x 16 =	80	GREEN	5
6 x 16 =	96	BLUE	6
7 x 16 =	112	YELLOW	7

Add together the values for the required combination and POKE it into the relevant location in character RAM.

Alternatively select the colour code for the bits set and multiply by 16 and add the code for the screen colour.

The above example would then be:-

Screen yellow = 16
Pixels black = 0
Value to be POKEd is 16 x 0 + 7 = 7

If you add line 35 below to your programme you will clear the screen to yellow and any points plotted will be in black.

The complete initialisation for bit mapping would now look like this:-

```
15 POKE53272,(PEEK(53272)OR8)
20 POKE51,255:POKE55,255:POKE52,31:POKE56,31
25 POKE53265,PEEK(53265)OR32
30 FORN=8192TO8192+8000:POKEN,0:NEXT
35 FORN=1024TO2023:POKEN,7:NEXT
```

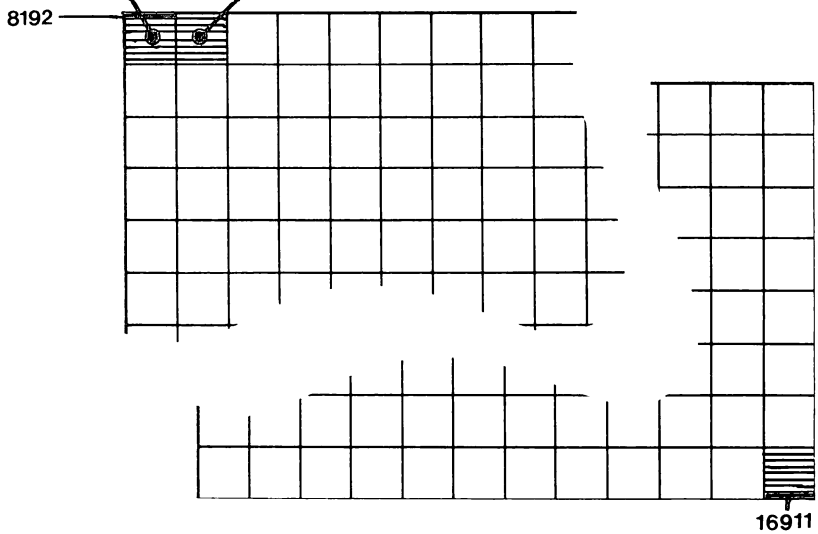
This process takes about 35 seconds to complete. A small machine code routine to accomplish this is shown below. In order to set the colour combinations you require POKE the value calculated as before into 247 and call the routine with SYS49152. It performs exactly the same functions as the Basic programme but in less than a second.

```
10 REM ************************************
15 REM * M-CODE BIT MAP ENABLE & CLEAR *
20 REM *   POKE 247,C TO SET COLOUR     *
25 REM * SYS 49152 TO SET UP AND CLEAR *
30 REM ************************************
50 IFPEEK(49152)<>8THENGOSUB50000
55 POKE247,7:SYS49152
9999 END
10000 DATA8,169,255,133,51,133,55,169,31
10005 DATA133,52,133,56,173,24,208,9,8,141
10010 DATA24,208,173,17,208,9,32,141,17,208
10015 DATA169,32,133,181,169,0,133,180,160
10020 DATA0,145,180,200,192,0,208,249,230
10025 DATA181,165,181,201,64,208,235,169,0
10030 DATA133,180,133,170,169,4,133,181,160
10035 DATA0,165,247,145,180,200,195,170,208
10040 DATA249,230,181,165,181,201,7,240,6
10045 DATA201,8,208,233,40,96,169,232,133
10050 DATA170,76,64,192
50000 FORN=0TO53
50005 READA
50010 POKEN+49152,A
50015 NEXT
50020 RETURN
```

Once the machine code has been loaded whenever you wish to clear the graphics screen POKE the colour combination you require into 247 and call the routine with SYS49152. You do not have to load the routine once it is in memory.

You are now ready to create bit mapped graphics.

8192	8200
8193	8201
8194	8202
8195	8203
8196	8204
8197	8205
8198	8206
8199	8207

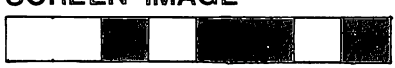


BIT PATTERN

0	0	1	0	1	1	0	1

(Decimal 45)

SCREEN IMAGE



The way in which the data held in the map memory is displayed on the screen does not allow you to directly specify X-Y coordinates. If you examine the manner in which the process takes place you will see why and develop a routine to allow X-Y information to be handled.

The start of the bit map is located at 8192. This corresponds to the start of the first row of pixels in the top left of the screen. Each bit of this location is mapped onto the screen as a pixel. Setting a bit to 1 will cause the pixel controlled by that bit to be displayed in the pixel colour which in the example is black.

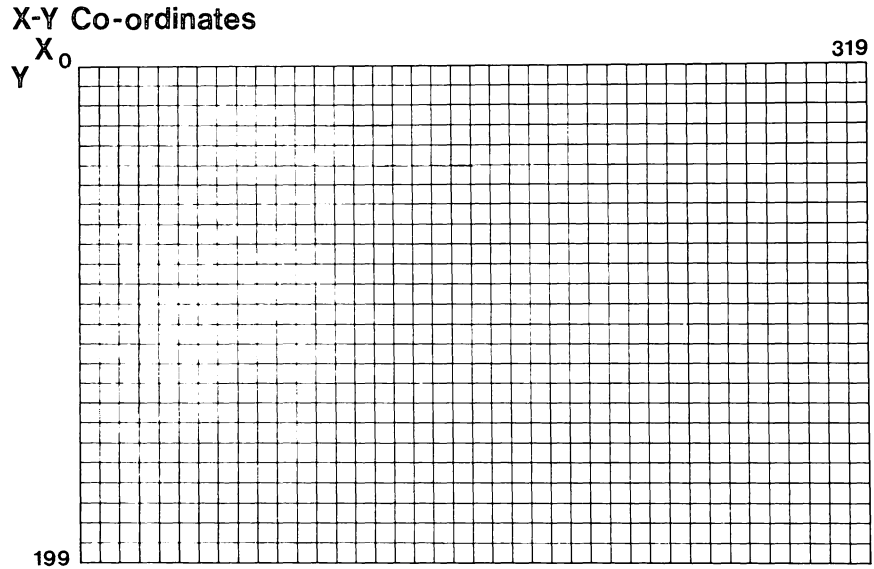
The second byte 8193 controls the pixel row below the first and so on until the 9th byte which controls the next row of pixels to the right.

As will be appreciated the manner in which the mapping occurs does not lend itself to a direct method of coordinate specification. In order to select whether a particular bit (or pixel as it appears on the screen) will be on or off you need to calculate not only the position on the screen but the location it is controlled by and finally which bit of that byte to alter.

Rather than leave you to struggle with this a small subroutine has been written that will enable you to enter the X-Y coordinates of a particular pixel and switch it on.

```
1000 A=INT(Y/8):B=8*((Y/8)-A)
1005 C=INT(X/8):PI=8*((X/8)-C):PI=7-PI
1010 LO=((A*320)+(C*8)+B)+8192
1015 POKELO,PEEK(LO)OR2↑PI
1020 RETURN
```

If this routine is incorporated in one of the previous initialisation programmes it becomes very simple to have full high resolution graphics facilities. All that is necessary is to add the appropriate lines to automatically calculate the coordinates and plot them on the screen by using GOSUB 1000.



The next programme will draw a square on the screen using FOR-NEXT loops to define the square.

```
10 REM ************************************
15 REM **          HIRES SQUARE          **
20 REM ************************************
25 REM
30 REM
50 IFPEEK(49152)<>8THENGOSUB30000
55 POKE247,7:SYS49152
100 Y=25:FORX=70TO250:GOSUB1000:NEXT
105 Y=175:FORX=70TO250:GOSUB1000:NEXT
110 X=70:FORY=25TO175:GOSUB1000:NEXT
115 X=250:FORY=25TO175:GOSUB1000:NEXT
```

```
120 GOTO120
1000 A=INT(Y/8):B=8*((Y/8)-A)
1005 C=INT(X/8):PI=8*((X/8)-C):PI=7-PI
1010 LO=((A*320)+(C*8)+B)+8192
1015 POKELO,PEEK(LO)OR2↑PI
1020 RETURN
10000 DATA8,169,255,133,51,133,55,169,31
10005 DATA133,52,133,56,173,24,208,9,8,141
10010 DATA24,208,173,17,208,9,32,141,17,208
10015 DATA169,32,133,181,169,0,133,180,160
10020 DATA0,145,180,200,192,0,208,249,230
10025 DATA181,165,181,201,64,208,235,169,0
10030 DATA133,180,133,170,169,4,133,181,160
10035 DATA0,165,247,145,180,200,196,170,208
10040 DATA249,230,181,165,181,201,7,240,6
10045 DATA201,8,208,233,40,96,169,232,133
10050 DATA170,76,64,192
50000 FORN=0TO95
50005 READA
50010 POKEN+49152,A
50015 NEXT
50020 RETURN
```

Line 50 checks to see if the machine code routine has been loaded. If it has then the programme passes to the next line. If the routine is not loaded it will GOSUB 50000 and load it.

Line 55 sets the colours used and clears the screen ready for the actual plotting.

Lines 100 to 115 create a series of coordinates that when plotted on the screen by the subroutine at line 1000 will create a square.

By using different routines at line 100 it is possible to generate an almost infinite variety of shapes including circles, elipses, spirals, curves etc.

Before moving on to more complex shapes, experiment further with the square to see what this small routine is capable of.

```
10 REM **********************************
15 REM **          MANY SQUARES          **
20 REM **********************************
25 REM
30 REM
50 IFPEEK(49152)<>8THENGOSUB50000
55 POKE247,7:SYS49152
100 YA=15:YB=185:XA=70:XB=250
105 FORJ=1TO8
110 YA=YA+10:YB=YB-10
115 XA=XA+10:XB=XB-10
120 Y=YA:FORX=XATOXB:GOSUB1000:NEXT
125 Y=YB:FORX=XATOXB:GOSUB1000:NEXT
130 X=XA:FORY=YATOYB:GOSUB1000:NEXT
135 X=XB:FORY=YATOYB:GOSUB1000:NEXT
140 NEXT
145 GOTO145
1000 A=INT(Y/8):B=8*((Y/8)-A)
1005 C=INT(X/8):PI=8*((X/8)-C):PI=7-PI
1010 LO=((A*320)+(C*8)+B)+8192
1015 POKELO,PEEK(LO)OR2↑PI
1020 RETURN
10000 DATA8,169,255,133,51,133,55,169,31
10005 DATA133,52,133,56,173,24,208,9,8,141
10010 DATA24,208,173,17,208,9,32,141,17,208
10015 DATA169,32,133,181,169,0,133,180,160
10020 DATA0,145,180,200,192,0,208,249,230
10025 DATA181,165,181,201,64,208,235,169,0
10030 DATA133,180,133,170,169,4,133,181,160
10035 DATA0,165,247,145,180,200,196,170,208
10040 DATA249,230,181,165,181,201,7,240,6
10045 DATA201,8,208,233,40,96,169,232,133
10050 DATA170,76,64,192
50000 FORN=0TO95
50005 READA
50010 POKEN+49152,A
50015 NEXT
50020 RETURN
```

The following examples will just show the lines that control the plotting routine. These should be placed in the programme above at line 100 onwards.

```
10 REM ********************************
15 REM **        HIRES CIRCLE          **
20 REM ********************************
25 REM
30 REM
100 XC=160:YC=100:R=80:XO=1:YO=0.8:A=2*π
105 N=500:I=A/N
110 FORV=0TOASTEPI
115 X=R*SIN(V):X=INT(X*XO+XC+4.99)
120 Y=R*COS(V):Y=INT(Y*YO+YC+4.99)
125 GOSUB1000
130 NEXT
135 GOTO135
```

XC and YC set the position of the circle. R is the radius and XO and YO are offset variables to determine the aspect ratio of the circle. For example by altering XO or YO an elipse can be drawn.

```
10 REM ********************************
15 REM **        HIRES ELIPSE          **
20 REM ********************************
25 REM
30 REM
100 XC=160:YC=95:R=80:XO=1:YO=1.2:A=2*π
105 N=500:I=A/N
110 FORV=0TOASTEPI
115 X=R*SIN(V):X=INT(X*XO+XC+4.99)
120 Y=R*COS(V):Y=INT(Y*YO+YC+4.99)
125 GOSUB1000
130 NEXT
135 GOTO135
```

By altering the value of the offset some very attractive patterns can be created.

```
10 REM ************************************
15 REM **              PATTERN             **
20 REM ************************************
25 REM
30 REM
100 XC=160:YC=95:R=40
105 FOROS=.2TO2STEP.2
110 XO=OS
115 YO=2-OS
120 FORV=0TO2*πSTEP.015
125 X=R*SIN(V):X=INT(X*XO+XC+.499)
130 Y=R*COS(V):Y=INT(Y*YO+YC+.499)
135 GOSUB1000
140 NEXT:NEXTOS
145 GOTO145
```

This example uses a FOR-NEXT loop to modify the offsets and thus produces a series of elipses. At this point it is worth noting that to obtain a true circle the offset should be 1.00. However due to the shape of the pixels displayed by the Commodore 64 a value of 0.8 gives a better result.

The final routine in this section will demonstrate that it is not necessary to use circles to generate striking effects.

```
10 REM ************************************
15 REM **          REID EX                 **
20 REM ************************************
25 REM
30 REM
50 IFPEEK(49152)<>8THENGOSUB50000
55 POKE53280,0:POKE247,32:SYS49152
100 YA=-8:YB=206:XA=-8:XB=322
105 FORJ=1TO13:YA=YA+8:YB=YB-8:XA=XA+8:XB=XB-8
110 X=XB+1:FORY=YATOYBSTEP((YB-YA)/(XB-XA))
115 X=X-1:GOSUB1000:NEXT
120 X=XA-1:FORY=YATOYBSTEP((YB-YA)/(XB-XA))
125 X=X+1:GOSUB1000:NEXT
130 NEXT
135 GOTO125
```

The biggest drawback with high resolution graphics routines written in Basic is that they are very slow to execute. Machine code will obviously overcome this problem but a detailed explanation of the necessary techniques is not possible in this book.

MULTI COLOUR BIT MAPPED GRAPHICS

Multi colour bit mapped graphics are also available on the Commodore 64. In the standard high resolution mode the maximum definition is 320 x 200 pixels. Since each pixel may be on or off this allows only two colours to be represented. Although it is possible to select the background colour and pixel colour for each 8 x 8 block it will not allow colour changes within that area.

In multi colour graphics each pixel is represented by two bits of memory. This allows the use of up to four independent colours for each pixel. The trade off is that because each pixel on the screen is mapped from two bits the pixel appears twice as wide thus restricting the horizontal resolution to 160 pixels. The vertical resolution of 200 pixels remains unchanged.

In order to select multi colour mode it is necessary to set the multi colour enable bit in the VIC II control register. This is located at 53270 and the enable bit is bit 4.

```
POKE53270,(PEEK(53270)OR16)
```

This assumes that you have already entered bit map mode.

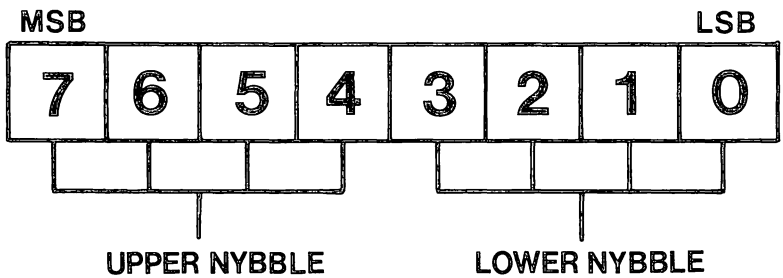
To disable the bit concerned

```
POKE53270,(PEEK(53270)AND239)
```

The correspondence between the bit pairs and the pixel colour are shown below:-

BITS	COLOUR	
00	SCREEN COLOUR	53281
01	UPPER SCREEN NYBBLE	1024-2023
10	LOWER SCREEN NYBBLE	1024-2023
11	COLOUR RAM	55296-56319

A nybble is 4 bits. Therefore there are 2 nybbles to a byte. The upper nybble is the 4 most significant bits and the lower nybble the 4 least significant.



To reiterate the colour of each pixel which is now mapped by two bits and appears twice as wide is determined as follows:-

00 colour as screen colour ie transparent
01 upper screen nybble selected by multiplying the normal colour code by 16
10 lower screen nybble the normal colour code for that colour added to the 01 colour value and POKEd to the screen area RAM
11 colour RAM colour code POKEd into the colour RAM

So to set the screen colour to white POKE53281,1 to set 01 to correspond to green and 10 to black 01 (upper nybble) = 5 x 16 + 0 (black) = 80 to set 11 to yellow the value is 7.

This routine will set the screen to white: ALL 01 pixels to green: ALL 10 pixels to black, and ALL 11 pixels to yellow. Since the information for each 8 x 8 block is held independently in the appropriate RAM area it is possible to determine the colour combination for each of the 8 x 8 squares.

```
10 REM ********************************
12 REM * MULTICOLOUR BIT MAPPING *
14 REM ********************************
16 REM
18 REM
20 POKE51,255:POKE52,31
25 POKE55,255:POKE56,31
30 REM ** RESERVE RAM FOR BIT MAP AREA
35 POKE53265,PEEK(53265)OR32
40 POKE53270,PEEK(53270)OR16
45 POKE53272,PEEK(53272)OR8
50 REM ** ENABLE M/COL BIT MAP
55 POKE53281,1:REM SET SCREEN (00) WHITE
60 FORN=1024TO2023
65 POKEN,80:NEXT
70 REM SET 01 PIXELS TO GREEN
75 REM SET 10 PIXELS TO BLACK
80 FORN=55296TO56295
85 POKEN,7:NEXT
90 REM SET 11 PIXELS TO YELLOW
95 FORN=0TO319:POKEN+8192,255:NEXT
100 FORN=320TO639:POKEN+8192,85:NEXT
105 FORN=640TO959:POKEN+8192,170:NEXT
110 FORN=960TO1279:POKEN+8192,0:NEXT
115 REM * DISPLAY '11' PIXELS IN 1ST ROW
120 REM * DISPLAY '01' PIXELS IN 2ND ROW
125 REM * DISPLAY '10' PIXELS IN 3RD ROW
130 REM * DISPLAY '00' PIXELS IN 4TH ROW
135 POKE53280,1
140 REM ** SET WHITE BORDER WHEN DONE
```

In the above example the first row on the screen is filled with 11 bit pairs causing the display to be in yellow.

The second row is in 01 pairs hence green.

The third row is in 10 pairs hence black.

The last row consists of 00 pairs and is 'transparent', the colour being the same as the screen colour, white.

The garbage at the bottom of the screen is caused by the contents of the bit map RAM area. This may be cleared in the same manner as that used for the bit map mode in the previous section.

In order to plot points and draw graphics on the multi colour screen in addition to the X-Y coordinates you have to specify which colour the point is to be plotted in.

The coordinates are now X 0 to 159 and Y 0 to 199.

The colour of each pixel is determined by the value of the variable CO.

```
10 REM ************************************
15 REM * MULTI COLOUR BIT MAP EXAMPLE *
20 REM ************************************
25 REM
30 REM
50 IFPEEK(49152)<>8THENGOSUB50000
55 POKE247,90:SYS49152
60 POKE53270,PEEK(53270)OR16
65 POKE53280,0:POKE53281,0
70 FORN=0TO999:POKE55296+N,6:NEXT
100 YA=1:YB=199:XA=1:XB=159
105 FORJ=1TO8
110 FORCO=1TO3
115 Y=YB:FORX=XATOXB:Y=Y-((YB-YA)/160)
120 GOSUB1000:NEXT
125 Y=YA:FORX=XATOXB:Y=Y+((YB-YA)/160)
130 GOSUB1000:NEXT
```

```
135 YA=YA+4:YB=YB-4
140 NEXT:NEXT
145 GOTO145
1000 XC=X*2
1005 A=INT(Y/8)
1007 B=8*((Y/8)-A)
1010 C=INT(XC/8)
1011 PI=8*((XC/8)-C):PI=6-PI
1012 PI=(2↑PI)*CO
1015 LO=((A*320)+(C*8)+B)+8192
1020 POKELO,PEEK(LO)ORPI
1025 RETURN
10000 DATA8,169,255,133,51,133,55,169,31
10005 DATA133,52,133,56,173,24,208,9,8,141
10010 DATA24,208,173,17,208,9,32,141,17,208
10015 DATA169,32,133,181,169,0,133,180,160
10020 DATA0,145,180,200,192,0,208,249,230
10025 DATA181,165,181,201,64,208,235,169,0
10030 DATA133,180,133,170,169,4,133,181,160
10035 DATA0,165,247,145,180,200,196,170,208
10040 DATA249,230,181,165,181,201,7,240,6
10045 DATA201,8,208,233,40,96,169,232,133
10050 DATA170,75,64,192
50000 FORN=0TO95
50005 READA
50010 POKEN+49152,A
50015 NEXT
50020 RETURN
```

The method employed to generate multi colour graphics is similar to that previously described. The main differences are that in multi colour mode you have to specify the colour the pixel will be displayed in and the X resolution is halved.

Finally remember to ensure that the colours chosen will be visible when they are displayed.

MOVING THE SCREEN AND VIC II CHIP

All the graphics facilities on the Commodore 64 are controlled by the VIC II chip. Whilst this is an extremely powerful microcircuit it does have one serious drawback. When using user defined characters and sprites the data has to be stored in the Basic area of memory thus limiting the space available for your programmes. This is because the VIC II chip can only 'see' or access memory in 16K blocks. The Commodore 64 has 64K of RAM which may be viewed as consisting of 4 sections of 16K. The block normally visible to the VIC chip is the first or block 0. This is also used by Basic. Normally the video controller does not need any RAM in this block with the exception of the 1000 bytes for the screen character memory. It is only when using bit mapping user defined characters or more than three sprites that you'lose' the precious Basic space.

However as the video controller can be made to 'look' at any 16K block within the Commodore 64 it is possible with a few POKEs to completely alter the situation.

The register that controls the position of the 16K block is located at 56576. The first two bits control the location of the 16K block which may have one of four positions.

Bank 0 is the normal position
Bank 1 starts at 16384
Bank 2 starts at 32768
Bank 3 starts at 49152

The character ROM image is only available in banks 0 and 2.

The positions of the screen character RAM and the position of user defined bit mapped and sprite graphics can also be modified.

Most of the time such modifications to the memory map cause a problem in another area of use.

For example if you move the VIC to bank 3 there are no ROM based characters available and half the space is used by the Basic interpreter.

Bearing in mind the needs of the graphics programmer who requires the maximum RAM for Basic and a reasonable space for user defined and other graphics information we would suggest that you make use of the re-location routine below.

```
10 REM ************************************
15 REM * MOVE VIC SCREEN AND CHAR DATA *
20 REM ************************************
30 REM *                                  *
35 REM * 100-105 LOWER BASIC              *
40 REM * 110-115 MOVE VIC TO BANK 2       *
45 REM * 120 MOVES SCREEN TO 35840        *
50 REM * 125 MOVES CHAR DATA TO 32768     *
55 REM * 130 TELLS OP.SYSTEM SCRN LOC     *
60 REM * 135-165 LOADS 1ST 128 CHARS      *
65 REM *                                  *
70 REM ************************************
100 POKE51,255:POKE52,127
105 POKE55,255:POKE56,127
110 POKE56578,PEEK(56578)OR3
115 POKE56576,((PEEK(56576)AND252)OR1)
120 POKE53272,((PEEK(53272)AND15)OR48)
125 POKE53272,(PEEK(53272)AND240)OR0
130 POKE648,140
135 POKE56334,PEEK(56334)AND254
140 POKE1,PEEK(1)AND251
145 FORN=0TO1023
150 POKE32768+N,PEEK(53248+N)
155 NEXT
160 POKE1,PEEK(1)OR4
165 POKE56334,PEEK(56334)OR1
```

The new locations will be:-

Screen memory from 1024-2023 to 35840-36839

Character data to 32768-34815 (RAM)

Sprite data area 34816-35839 (or character data)

Sprite pointers from 2040-2047 to 36856-36863

The colour RAM remains at 55296-56295

The programme will also download the first 128 ROM characters to give normal upper case and some graphics. The RAM area 32768 to 34815 will hold the 128 downloaded characters and up to 128 user defined characters. If you are using sprites the area from 24816 to 35839 will hold up to 16 sets of sprite data. If you are not using sprites this area may be used for further user defined character storage.

Once the changes are made the computer will not be able to run programmes that were written for the standard configuration if they make use of POKEs to the screen RAM or store user defined character data in the normal locations.

Basic finishes at 32767 which will give the user a total of 30K of RAM for programmes. This together will the storage for graphic data will enable the programmer to write far more complex programmes than would be possible with the standard configuration.

JOYSTICKS

The Commodore 64 allows the use of two joysticks. Each joystick is connected to an 8 bit PORT. The port can be viewed as being a location in memory.

To read a joystick value ie to determine its position you simply PEEK the relevant port location.

Because each port is wired in a slightly different manner the routine to convert the value obtained into a usable form is also different.

At this point it would be worthwhile to explain how a joystick works.

There are five switches in the joystick corresponding to UP DOWN LEFT RIGHT and FIRE. Each switch controls one bit of the port. The connections are shown below:-

```
7 6 5 4 3 2 1 0        PORT 1 connections
0 0 0 0 0 0 0 0        location 56321
X X X f r l d u      X = unused (set to 1)
                     f = fire u = up r = right
                      d = down l = left
```

Each bit is usually set to logical 1 so the value with the stick in the centre is 128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = 255. Each time a switch is closed ie when the joystick is moved the corresponding bit is set to logical 0.

As it is easier to start with the centre position at 0 it is necessary to convert the PEEK value to a form that is readily understandable. This is done by subtracting the PEEK from 255.

For example to read a joystick in port 1.

```
10 REM ** JOYSTICK POSITION **
40 PRINT"□"
50 J1=PEEK(56321)
60 J1=255-J1
70 PRINT"■JOYSTICK POSITION VALUE IS ";J1
80 GOTO50
```

If you run the programme and change the position of the joystick you will be able to see how the value changes.

In the above example the variable J has been used to hold the combined position of the joystick. However it is far more useful to be able to isolate each position. This is done by using a logical AND (if you are unfamiliar with AND don't worry it will work anyway).

```
10 REM ** JOYSTICK READ **
1000 JS=255-PEEK(56321):REM READ PORT 1
1005 IF JS AND 1 THEN A$= "UP"
1006 REM IF BIT 0 SET JOYSTICK IS UP
1010 IF JS AND 2 THEN A$= "DOWN"
1011 REM IF BIT 1 SET JOYSTICK IS DOWN
1015 IF JS AND 4 THEN A$= "LEFT"
1016 REM IF BIT 2 SET JOYSTICK IS LEFT
1020 IF JS AND 8 THEN A$= "RIGHT"
1021 REM IF BIT 3 SET JOYSTICK IS RIGHT
1025 IF JS AND 16 THEN A$= "FIRE"
1026 REM IF BIT 4 SET JOYSTICK IS FIRING
1030 IF JS = 0 THEN A$="CENTRE"
1031 REM IF NONE SET JOYSTICK IS CENTRED
1035 PRINT"□JOYSTICK POSTION =";A$
1040 GOTO1000
```

The above programme has one big flaw. It will not respond to combined positions ie UP and FIRE.

The next routine overcomes this.

```
10 REM ** JOYSTICK COMBINED **
100 A=255-PEEK(56321)
110 IFAANDITHENA$="UP "
120 IFAANDTHENA$=A$+"DOWN "
130 IFAANDTHENA$=A$+"LEFT "
140 IFAANDTHENA$=A$+"RIGHT "
150 IFAANDTHENA$=A$+"FIRE "
160 IFA$=""THENA$="CENTRE"
170 PRINT"JOY 1 ";A$:A$=""
180 GOTO100
```

Enter the above programme to check that the screen displays the expected position.

In order to make use of the information on the joystick position it is easier to assign a vari-able to each position and test them individually in the main programme. The following is an example of a subroutine to read the joystick position and return the variable accordingly set. The reader may wish to use a different method for a particular application but this will provide enough detail to enable you to write your own versions.

```
10 REM ** USER JOY ONE **
100 GOSUB1000
105 PRINT"UP=";UP
110 PRINT"DN=";DN
115 PRINT"LF=";LF
120 PRINT"RT=";RT
125 PRINT"FR=";FR
130 GOTO100
1000 UP=0:DN=0:LF=0:RT=0:FR=0
1010 A=255-PEEK(56321)
1020 IFA AND 1 THEN UP=1
1030 IFA AND 2 THEN DN=1
1040 IFA AND 4 THEN LF=1
1050 IFA AND 8 THEN RT=1
1060 IFA AND 16 THEN FR=1
1070 RETURN
```

In this example five variables are set according

to the position of the joystick at port 1.

UP is set to 1 if joystick is up
DN is set to 1 if joystick is down
LF is set to 1 if joystick is left
RT is set to 1 if joystick is right
FR is set to 1 if the fire button is pressed

As mentioned earlier, the routine for joystick 2 at port 2, is slightly different.

The location of the port is 56320. The difference is that instead of all bits being set to 1 if no switch is closed bit 7 is set to 0. To overcome this simply subtract the PEEK from 127.

The modified form of the above programme is shown below.

```
10 REM ** USER JOY TWO **
100 GOSUB1000
105 PRINT"UP=";UP
110 PRINT"DN=";DN
115 PRINT"LF=";LF
120 PRINT"RT=";RT
125 PRINT"FR=";FR
130 GOTO100
1000 UP=0:DN=0:LF=0:RT=0:FR=0
1010 A=127-PEEK(56320)
1020 IFA AND 1 THEN UP=1
1030 IFA AND 2 THEN DN=1
1040 IFA AND 4 THEN LF=1
1050 IFA AND 8 THEN RT=1
1060 IFA AND 16 THEN FR=1
1070 RETURN
```

Both programmes may be combined to read both joysticks as shown below.

```
10 REM ** USER BOTH JOYS **
50 PRINT"□"
100 GOSUB1000
105 PRINT"■JOY 1 UP=";UP(1)
110 PRINT"JOY 1 DN=";DN(1)
115 PRINT"JOY 1 LF=";LF(1)
120 PRINT"JOY 1 RT=";RT(1)
125 PRINT"JOY 1 FR=";FR(1)
200 PRINT"■JOY 2 UP=";UP(2)
205 PRINT"JOY 2 DN=";DN(2)
210 PRINT"JOY 2 LF=";LF(2)
215 PRINT"JOY 2 RT=";RT(2)
220 PRINT"JOY 2 FR=";FR(2)
225 GOTO100
1000 UP(1)=0:DN(1)=0:LF(1)=0:RT(1)=0:FR(1)=0
1010 A=255-PEEK(56321)
1020 IFA AND 1 THEN UP(1)=1
1030 IFA AND 2 THEN DN(1)=1
1040 IFA AND 4 THEN LF(1)=1
1050 IFA AND 8 THEN RT(1)=1
1060 IFA AND 16 THEN FR(1)=1
2000 UP(2)=0:DN(2)=0:LF(2)=0:RT(2)=0:FR(2)=0
2010 A=127-PEEK(56320)
2020 IFA AND 1 THEN UP(2)=1
2030 IFA AND 2 THEN DN(2)=1
2040 IFA AND 4 THEN LF(2)=1
2050 IFA AND 8 THEN RT(2)=1
2060 IFA AND 16 THEN FR(2)=1
2070 RETURN
```

One very powerful combination is the ability to use joysticks in graphic design. Although programmes that exploit the possibilities to the greatest extent are complex and lengthy it is quite feasible to demonstrate the capabilities using a fairly simple programme.

```
10 REM ********************************
15 REM * JOYSTICK BIT MAPPED GRAPHICS *
20 REM ********************************
25 REM
30 REM
50 POKE51,255:POKE52,31:POKE55,255:POKE56,31
55 IFPEEK(49152)<>8THENGOSUB50000
60 POKE247,80:SYS49152
65 X=160:Y=100:GOSUB1000
100 A=255-PEEK(56321):XC=0:YC=0
105 IFAANDITHENYC=-1
110 IFAANDZTHENYC=+1
115 IFAANDYTHENXC=-1
120 IFAANDXTHENXC=+1
125 IFAANDMTHENFB=1
130 X=X+XC:Y=Y+YC
135 IFX<1THENX=0:GOTO100
140 IFX>319THENX=319:GOTO100
145 IFY<1THENY=1:GOTO100
150 IFY>199THENY=199:GOTO100
155 GOSUB1000
160 IFFB=1THENFB=0:GOTO200
165 GOTO100
200 XB=X-XC:YB=Y-YC
205 A=INT(YB/8):B=8*((YB/8)-A)
210 C=INT(XB/8):P=8*((XB/8)-C):P=7-P
215 BY=((A*320)+(C*8)+B)+8192
220 POKEBY,PEEK(BY)AND(255-2↑P)
225 GOTO100
1000 A=INT(Y/8):B=8*((Y/8)-A)
1005 C=INT(X/8):P=8*((X/8)-C):P=7-P
1010 BY=((A*320)+(C*8)+B)+8192
1015 POKEBY,PEEK(BY)OR2↑P
1020 RETURN
10000 DATA8,169,255,133,51,133,55,169,31
10005 DATA133,52,133,56,173,24,208,9,8,141
10010 DATA24,208,173,17,208,9,32,141,17,208
10015 DATA169,32,133,181,169,0,133,180,160
10020 DATA0,145,180,200,192,0,208,249,230
10025 DATA181,165,181,201,64,208,235,169,0
10030 DATA133,180,133,170,169,4,133,181,160
10035 DATA0,165,247,145,180,200,196,170,208
```

```
10040 DATA249,230,181,165,181,201,7,240,6
10045 DATA201,8,208,233,40,96,169,232,133
10050 DATA170,76,64,192
50000 FORN=0TO95
50005 READA
50010 POKEN+49152,A
50015 NEXT
50020 RETURN
```

In this example a joystick connected to port 1 controls the movement of a single high resolution pixel across the screen. Movement of the joystick will cause the dot to trace a line in the direction of the joystick. If you depress the fire button the dot will flash. If the fire button is pressed and the stick moved the dot will move without tracing a line and if the dot crosses a line already drawn it will erase it.

The programme is straight-forward and should hold no mysteries if the chapter on bit mapping has been read thoroughly.

The only addition to the techniques discussed so far is the "unplotting" routine at lines 200 to 255. The coordinates calculated by the joystick reading routine at lines 100 to 130 are translated into the pixel position and the pixel removed by ANDing the value with the contents of the memory location immediately before in the direction of the last movement.

The colour of the screen and pixel plot may be modified in a similar manner to that already described in the section on bit mapping by changing the number POKEd into location 247 in line 50.

SOUND AND MUSIC

The enormous potential for generating sound on the Commodore 64 is due to the Sound Interface Device (SID) chip, one of the most complex parts of the computer.

By means of a series of registers within the SID chip it is possible to create a wide variety of sound from zapps and explosions to very pleasing music.

This chapter will deal with ways to generate special sound effects and play music.

In order to fully understand how to create a specific sound it is necessary to understand the general principles of waveform volume attack-decay sustain-release and pitch.

The registers in the SID chip are responsible for controlling all these features and a few more besides.

The steps involved in making sound may be summarized:-

1. Overall volume (loudness)
2. Envelope shape (attack-decay sustain-release)
3. Pitch
4. Waveform

There are three separate oscillators available each capable of separate control. This enables sounds from each to be 'mixed' to obtain chord-like sounds or even to use one oscillator to control another and thus widen the range of effects available.

Before tackling advanced features we shall examine the principles behind sound generation.

1. VOLUME

The volume or loudness is controlled by a register in the SID chip. This is located at 54296 and will set the overall volume for all of the oscillators available. The volume may be set by POKEing a number between 0 (off) and 15 (highest) into the register.

```
POKE54296,15
```

will thus set the volume for ALL sound output to the maximum level and

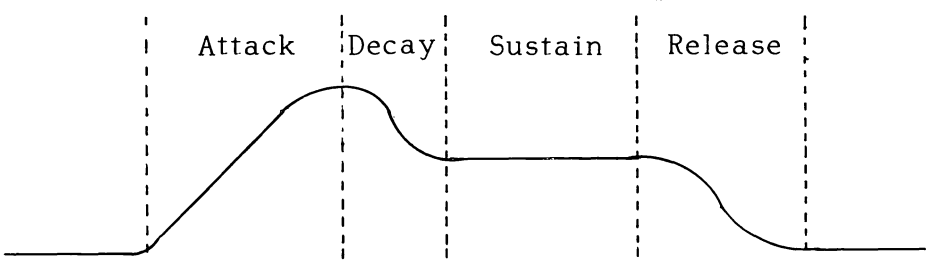
```
POKE54296,0
```

will turn it off.

In all the examples in this book the volume is set to 15 and you will be able to control the actual volume level by means of the control on your TV monitor.

2. ATTACK-DECAY SUSTAIN-RELEASE

Attack decay sustain release (ADSR) control the ENVELOPE the sound is formed in. When a note is produced its volume builds up to the maximum level and dies away. The manner in which the note changes is known as the ENVELOPE and the four parameters of the envelope attack-decay and sustain-release are all programmable characteristics. The diagram below will illustrate the envelope concept.



ATTACK is the build up from silence to what is normally the loudest part of the note. DECAY is the transition from the peak to the steady state volume of sustain. SUSTAIN is the period where the note remains virtually constant up to the time RELEASE occurs. The total shape of the ADSR curve is the ENVELOPE and has a considerable influence on what the note actually sounds like.

For example a piano note has a relatively short attack-decay phase, a short sustain and a long slow release. By contrast an organ produces a note that is relatively slow to build up with a small decay to a long sustain and a long release.

The SID chip will allow you to modify the overall envelope with registers that control all phases of the note.

Each oscillator has two locations that control attack-decay and sustain-release.

They are located at:-

OSCILLATOR	ATTACK/DECAY	SUSTAIN/RELEASE
1	54277	54278
2	54284	54285
3	54291	54292

Each register is divided into two parts . The first part consists of bits 7-4 and the second from 3-0.

In the attack-decay byte the attack is controlled by the first part bits 7-4 and the release by the second bits 3-0.

Since each half is made up from 4 bits the value of each half may be between 0 and 15.

The sustain-release byte is split in a similar fashion with the first 4 bits controlling the sustain and the second 4 the release.

The approximate duration for each value is shown below:-

VALUE	ATTACK PERIOD	DECAY AND RELEASE
0	2 mS	6 mS*
1	8	24
2	16	48
3	24	72
4	38	114
5	56	168
6	68	204
7	80	240
8	100	300
9	250	750
10	500	1500 (1.5 S)
11	800	2400 (2.4 S)
12	1000 (1 S)	3000 (3 S)
13	3000 (3 S)	9000 (9 S)
14	5000 (5 S)	15000 (15 S)
15	8000 (8 S)	24000 (24 S)

* = milliseconds 1 millisecond = 1000th second

The sustain value sets the level of the VOLUME for the duration of the note. Thus a value of 15 would cause the output from that oscillator to be at its highest and a level of 0 would set it to the lowest level.

To set the envelope characteristics to those desired you first need to calculate the combined value to be POKEd into the appropriate registers.

Since the first half of each register is composed of the most significant 4 bits the value calculated for for the attack and sustain values must be multiplied by 16 and added to the decay-release figures.

For example to set the parameters as follows:-

OSCILLATOR 1

ATTACK:	80 mS	7
DECAY:	72 mS	3
SUSTAIN:	Max volume	15
RELEASE:	204 mS	6

Values:	ATTACK	7
	DECAY	3
	SUSTAIN	15
	RELEASE	6

Combined attack/decay = 7 x 16 + 3 = 115
Combined sustain/release = 15 x 16 + 6 = 246

```
POKE54277,115
POKE54278,246
```

The sustain value only controls the maximum volume not the duration of the note. The usual method of controlling the length of time a note is played is to turn the oscillator on by POKEing the waveform value in and silencing the note by turning it off after the required length of time. One method is shown below:-

```
POKE54276,17 turn on voice 1 triangle
For T = 1 TO 500:NEXT duration of note approximately ½ second.
POKE54276,16 turn off voice 1 triangle
```

The RELEASE time will be added to the duration as set by the ADSR parameters.

The lifetime of a single note can now be viewed thus:-

The note starts when the waveform is POKEd into the appropriate register.

ATTACK VALUE sets the time over which the note builds to its peak.

DECAY VALUE determines the delay before the note falls to the volume set by the SUSTAIN.

The actual length of the note is determined by the time delay between turning the waveform on and turning it off.

Once the note has been turned off the RELEASE value determines the time the note will take to reach zero volume.

3. PITCH

The pitch or note of the oscillator is controlled by 2 registers for each voice.

This allows an effective range of 0 to 65535 different notes thus enabling a very accurate control of the pitch produced.

The locations are:-

OSCILLATOR	HIGH BYTE	LOW BYTE
1	54273	54272
2	54280	54279
3.	54287	54286

The value to be POKEd into each register may be calculated thus:-

HIGH BYTE = INT(VALUE÷256)
LOW BYTE = VALUE-(INT(VALUE÷256))x256

where VALUE is the value of the note required.

The following table gives the values for two octaves starting at Middle C.

NOTE	VALUE	HIGH	LOW
C-4	4291	16	195
C#	4547	17	195
D-4	4817	18	209
D#	5103	19	239
E-4	5407	21	31
F-4	5728	22	96
F#	6069	23	181
G-4	6430	25	30
G#	6812	26	156
A-4	7217	28	49
A#	7647	29	223
B-4	8101	31	165
C-5	8583	33	135
C#	9094	35	134
D-5	9634	37	162
D#	10207	39	223
E-5	10814	42	62
F-5	11457	44	103
F#	12139	47	107
G-5	12860	50	60
G#	13626	53	57
A-5	14435	56	99
A#	15294	59	190
B-5	16203	63	75

4. WAVEFORM

Each oscillator is capable of producing four different waveforms, one at any given moment.

They are:-

1. Triangle
2. Sawtooth
3. Pulse
4. Random (noise)

Each waveform will affect the quality of the sound generated and may be modified to suit the particular type of tonal quality desired.

The waveform generated is controlled by 3 registers, one for each oscillator. They are located at:-

OSCILLATOR 1 54276
OSCILLATOR 2 54283
OSCILLATOR 3 54290

The waveform is changed by a POKE

POKE 54276,17 sets osc. 1 to TRIANGLE
POKE 54276,33 sets osc. 1 to SAWTOOTH
POKE 54276,65 sets osc. 1 to PULSE
POKE54276,129 sets osc. 1 to NOISE

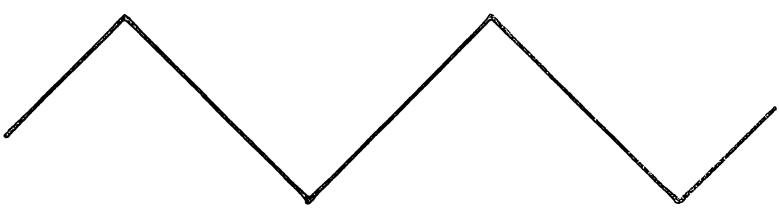
Selecting the waveform is the last operation required to make a sound audible. When the volume ADSR and pitch of the note have been selected the oscillator is turned on by setting the waveform to the desired shape and the note will play until the waveform is cleared by one of the following POKEs

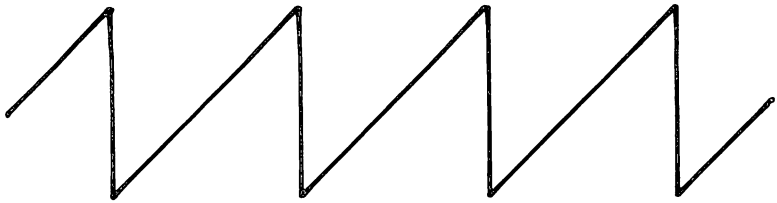
OSCILLATOR 1

POKE54276,16 turn off TRIANGLE
POKE54276,3 2 turn off SAWTOOTH
POKE54276,64 turn off PULSE
POKE54276,128 turn off NOISE

As soon as the waveform is turned off the RELEASE will come into operation to complete the envelope shape.

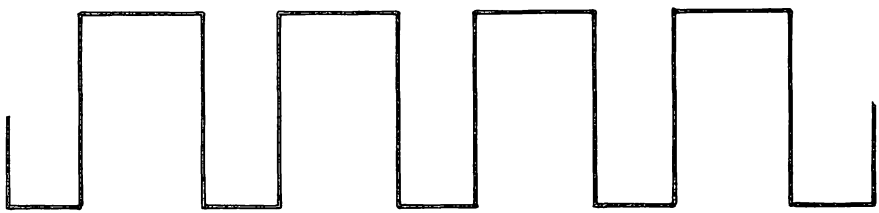
TRIANGLE WAVEFORM





SAWTOOTH WAVEFORM

The pulse shape is effectivley a square wave.



The pulse width is variable and must be set by the user before using this waveform. The details are given later in this chapter.

The last waveform is noise and as suggested by the name creates a random waveform that sounds like escaping steam.

This is normally used to create sound effects.

The registers that control the pulse width are:-

OSCILLATOR	LOW BYTE	HIGH NYBBLE
1	54274	54275
2	54281	54282
3	54288	54289

The value is thus composed of a 12 bit number allowing a range of 0 to 4095.

The pulse width may be calculated with this formula:-

WIDTH = (VALUE/4095) x 100

where the width is expressed as a percentage of one cycle.

A VALUE of 2048 will therefore give a 50% duty cycle or a true square wave.

A VALUE of 1024 will give a 25% duty cycle where the pulse ON time is 25% of the cycle duration.

A VALUE of 3072 will give a 75% duty cycle and so forth.

The formula for calculating the numbers to be POKEd is given below:-

HIGH = INT(VALUE/256)
LOW = VALUE - (INT(VALUE ÷ 256)) x 256

The SID chip will also allow the sound produced to be modified in other ways. Before moving on to the more complex aspects of sound modification we shall examine a few of the concepts already described in more detail.

Any programme using sound must follow certain principles in order for it to work. These are summarized below:-

1. Set the overall volume by POKEing 54272 with a number between 0 (off) and 15 (loudest).

2. Set the envelope for each voice in use by POKEing the appropriate values into the attack-decay and sustain-release registers.

3. Set the pitch of each note to be played.

4. Select the waveform for each oscillator.

When programming sound effects the actual pitch of the note need not and in some cases cannot be set to any normal musical note. Because of this programming sounds rather than music as such is a little easier. We will now examine some of the techniques involved in generating some useful sound effects.

SOUND EFFECTS PROGRAMMING

Virtually all games programmes use sound to enhance their appeal. The sound effects most employed are probably the explosion 'zapp' and an 'approaching' noise similar to footsteps. Since any or all of these effects may be required by different parts of the programme it would be sensible to structure the programmes as subroutines that may be called by various sections of the main programme.

The example given below will produce an 'explosion' when called by GOSUB5000. The first lines set up the volume and clear the SID chip registers and should be placed near the start of the main programme to ensure they are only executed once.

```
50 FORN=54272TO54296:POKEN,0:NEXT
60 POKE54296,15
100 GOSUB5000:END
5000 POKE54277,25
5005 POKE54276,129
5010 POKE54278,253
5015 POKE54273,7
5020 POKE54272,128
5025 POKE54276,128
5030 RETURN
```

5000 sets the attack-decay
5005 sets the waveform to NOISE
5010 sets the sustain-release
5015 sets the High byte for the frequency
5020 sets the Low byte for the frequency
5025 turns the waveform off
5030 returns to the main programme

An interesting feature of this routine is that the duration of the sound is controlled by the RELEASE setting. This is so that the effect may continue without interrupting any other action, eg graphics that may be occuring.

Although the sound produced is not really much like an explosion with suitable graphics the result is very effective.

```
50 FORN=54272TO54296:POKEN,0:NEXT
60 POKE54296,15
100 GOSUB5000:END
5000 POKE54277,25 :REM ** ATTACK/DECAY
5005 POKE54278,246:REM ** SUST/RELEASE
5010 POKE54272,12: REM ** LOW PITCH
5015 POKE54276,129:REM ** NOISE ON
5020 FORN=10TO60
5025 POKE54273,N  :REM ** HIGH PITCH
5030 NEXT
5035 POKE54276,128:REM ** NOISE OFF
5040 RETURN
```

5000 sets the attack-decay
5005 sets the sustain-release
5010 sets the low byte for the frequency
5025 turns on the noise waveform for osc. 1.
5020 starts a loop to modify the frequency of osc. 1
5025 changes the frequency of the noise
5030 increments the loop
5035 turns the sound off
5040 returns to the main programme

The main difference in this routine is in the use of a FOR-NEXT loop to modify the sound whilst it is still audible. This is responsible for the overall sound produced.

```
50 FORN=54272TO54296:POKEN,0:NEXT
60 POKE54296,15
100 FORX=1TO10:GOSUB5000
105 FORT=1TO700:NEXT
110 NEXT:END
5000 POKE54277,25 :REM ** ATTACK/DECAY
5005 POKE54278,244:REM ** SUST/RELEASE
5010 POKE54272,128:REM ** LOW PITCH
5015 POKE54273,20 :REM ** HIGH PITCH
5025 POKE54276,129:REM ** NOISE ON
5030 FORT=1TO50:NEXT
5035 POKE54276,128:REM ** NOISE OFF
5040 RETURN
```

This will make the sort of sound that may be used to indicate the approach of an 'alien' or other object.

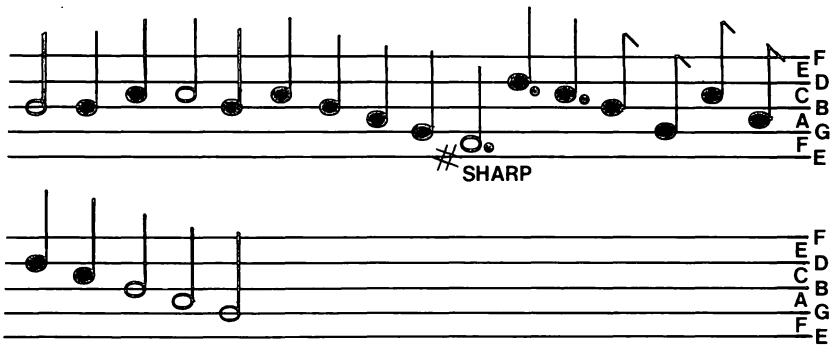
MUSIC

The sound generators in the Commodore 64 may also be employed to play music. The fundamental principles of sound formation have already been examined and this section will explore the ways of turning the noises produced into music.

The two most fundamental aspects of music are the pitch of the note and the time for which it is played.

In musical notation the pitch of the note is represented by its position on the stave and the duration by the shape of the notes.

An example is shown below:-



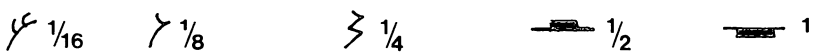
The various length notes are named thus:-

APPEARANCE	NAME	DURATION	VALUE
	demi-semi-quaver	1/32	0.5
	semi-quaver	1/16	1
	quaver	1/8	2
	crotchet	1/4	4
	minim	1/2	8
	semibreve	1	16

In addition a note may be written with a dot beside it. This indicates that it should be played for $1\frac{1}{2}$ times the usual time.

Thus a dotted quaver  would be played for 3/16.

There are times when no note is played. These are called rests and again there are symbols to represent the length of the pause:-



In order to translate music to be played by the computer you need to convert the notes and lengths into data.

Using the example shown before:-

NOTE	VALUE	NAME	TIME	DURATION/ VALUE
B-4	8101	minim	1/2	8
B-4	8101	crotchet	1/4	4
C-5	8583	crotchet	1/4	4
C-5	8583	minim	1/2	8
B-4	8101	crotchet	1/4	4
C-5	8583	crotchet	1/4	4
B-4	8101	crotchet	1/4	4
A-4	7217	crotchet	1/4	4
G-4	6430	crotchet	1/4	4
F#4	6069	minim dot	3/4	12
D-5	9634	crotch.dot	3/8	6
C-5	8583	crotch.dot	3/8	6
B-4	8101	quaver	1/8	2
G-4	6430	quaver	1/8	2
C-5	8583	quaver	1/8	2
A-4	7217	quaver	1/8	2
D-5	9634	crotchet	1/4	4
C-5	8583	crotchet	1/4	4
B-4	8101	minim	1/2	8
A-4	7217	minim	1/2	8
G-4	6430	minim	1/2	8

The DURATION value is used to determine how long the note will be played for.

When a rest occurs the VALUE ie pitch of note is set to 0 and the DURATION value set according to the table above.

Once the music has been translated into number form it can be incorporated into DATA statements.

```
10 REM ********************************
15 REM ** MUSIC EXAMPLE SAWTOOTH **
20 REM ********************************
25 REM
30 REM
50 FORN=54272TO54296:POKEN,0:NEXT
100 POKE54296,15  :REM ** VOLUME
105 POKE54277,115 :REM ** ATTACK/DECAY
110 POKE54278,246 :REM ** SUST/RELEASE
115 READF,D:PRINTF,D
120 IFD=0THENEND
125 HF=INT(F/256)
130 LF=F-(INT(F/256)*256)
135 POKE54272,LF:POKE54273,HF
140 POKE54276,33 :REM ** WAVEFORM
145 FORT=1TOD*64:NEXT
150 POKE54276,32 :REM ** WAVEFORM
155 GOTO115
500 DATA8101,8,8101,4,8583,4
505 DATA8583,8,8101,4,8583,4
510 DATA8101,4,7217,4,6430,4
515 DATA6069,12,9634,6,8583,6
520 DATA8101,2,6430,2,8583,2
525 DATA7217,2,9634,4,8583,4
530 DATA8101,8,7217,8,6430,8
540 DATA0,0
```

50 clears the SID chip registers
100 sets the volume to maximum
105 sets the attack-decay
110 sets the sustain-release
115 reads the values for the note and duration into variables F (frequency) and D (duration)
120 checks to see if the tune has been played
125 calculates the high byte for the note
130 calculates the low byte
135 POKEs the high and low byte into voice 1 frequency registers
140 turns the sound on
145 uses the delay value to determine the duration of the note
150 turns the note off
155 loops back to play the next note

500-530 contain the data for the note and duration, 540 holds 2 zero's to indicate that all the music has been played

By using different settings for the generation of the sound a wide variety of pleasing tones may be produced.

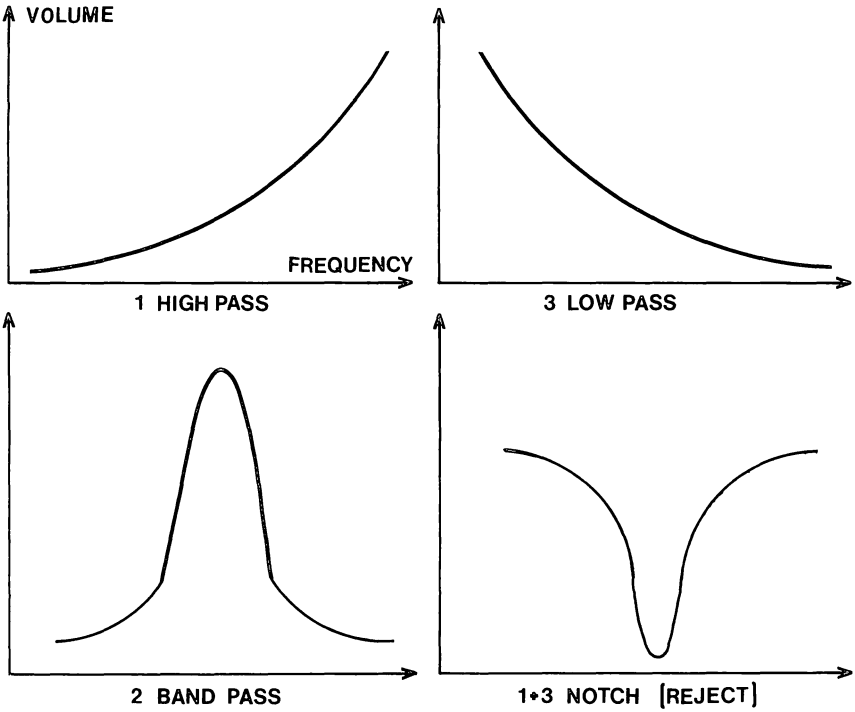
For example the changes below will cause the sound to imitate a guitar.

```
50 FORN=54272TO54296:POKEN,0:NEXT
100 POKE54296,15  :REM ** VOLUME
105 POKE54277,10  :REM ** ATTACK/DECAY
110 POKE54278,0   :REM ** SUST/RELEASE
115 READF,D:PRINTF,D
120 IFD=0THENEND
125 HF=INT(F/256)
130 LF=F-(INT(F/256)*256)
135 POKE54272,LF:POKE54273,HF
140 POKE54276,33 :REM ** WAVEFORM
145 FORT=1TOD*64:NEXT
150 POKE54276,32 :REM ** WAVEFORM
155 GOTO115
```

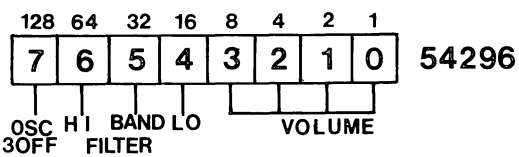
One method of modifying the sound produced is by filtering.

The filters available on the Commodore 64 allow you to select the frequency at which the filter will operate and the type of filter to be used.

There are three basis types of filter and their operation may be combined to form others



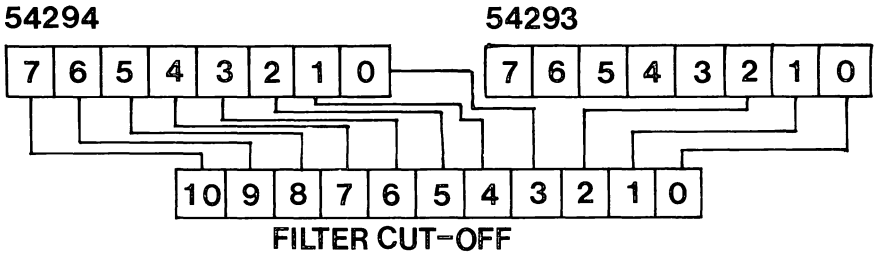
Each filter is controlled by a bit in the same register that controls the overall volume 54296.



Assuming that you want to keep the volume set to maximum you can select the filter mode in the following way:-

LOW PASS: POKE54296,31
BAND PASS: POKE54296,47
HIGH PASS: POKE54296,79
NOTCH: POKE54296,95

The cut-off frequency of the selected filter must also be set. This is held as an 11 bit number at locations 54293 and 54294.



As can be seen 54293 holds the lowest 3 bits in positions 0-2 and 54294 holds the 8 most significant bits.

The cut-off frequency may be modified over a range of approximately 1KHz to 12KHz by selecting the appropriate values to be POKEd into the above registers.

Location 54295 contains the bits that control which oscillator's output will be fed through the filter together with 4 bits that control the resonance. Resonance is a feature that emphasizes the frequency component at the frequency selected for cut-off when the filter is employed and has a value between 0 (no resonance) and 15 (maximum).

All three voices may be routed through the filter if desired.

To route an oscillator through the filter all that is necessary is to set the relevant bit in the register.

```
POKE54295,1     for osc 1 and no resonance
POKE54295,241   for osc 1 and maximum resonance
POKE54295,2     for osc 2 and no resonance
POKE54295,242   for osc 2 and maximum resonance
POKE54295,4     for osc 3 and no resonance
POKE54295,244   for osc 3 and maximum resonance
```

To send more than 1 voice through the filter add together the values for each oscillator to be routed:-

POKE54295,3 will route oscillators 1 and 2 through the filter with no resonance etc.

Ring modulation is also possible where the output from one oscillator is used to control another. The POKEs required are shown below together with the POKEs to synchronise one oscillator with another.

RING MODULATE osc 1 with 3

Add 4 to the WAVEFORM VALUE and POKE it into 54276

OSC 2 with 1

Add 4 to the WAVEFORM VALUE and POKE it into 54283

OSC 3 with OSC 2

Add 4 to the WAVEFORM VALUE and POKE it into 54290

SYNC OSC 1 with 3 add 2 to the waveform and ring modulation figure and POKE into 54276

SYNC OSC 2 with OSC 1 add 2 to the waveform and ring modulation figure and POKE it into 54283

SYNC OSC 3 with OSC 2 add 2 to the waveform and ring modulation figure and POKE it into 54290

It is impossible to give more than a guide to the sound capabilities of the Commodore 64 and the best way to learn and use the enormous range of facilities is by experiment. It is hoped that this brief introduction to the SID chip will provide enough raw material to allow the user to experiment further for example by altering the pitch of an oscillator whilst it is on and dynamically changing the filtering.

DISK DRIVES

Probably the most useful peripheral that may be connected to the Commodore 64 is the disk drive. Each VIC 1541 single floppy disk is able to hold a maximum of 169,000 bytes of information and accesses the data in a fraction of the time taken by a cassette deck.

The Commodore 64 will also permit the connection of two three or more disk drives to increase the storage capacity even further.

The disk drive is normally accessed as device 8 eg

```
LOAD"MARTIAN HOP",8
```

would look for and load (if it was on the disk) a programme called "MARTIAN HOP" from device 8 the disk drive.

When using more than one disk it is necessary to alter the device number of each additional disk drive. For example if two disk drives were in use it would make sense to call the first with device 8 and the second with device 9 etc.

There are two ways in which to modify the device number the first involves removing the disk drive from its case and cutting jumpers. This is NOT recommended unless you are aware of the risks involved and should not be attempted by anyone other than a qualified electronic engineer.

The second method uses software to achieve the same end. The only disadvantage of the "soft" change is that the routine must be repeated every time the drive is switched on.

The short programme here will allow you to change the device number as wished.

```
10 REM **** DISK DEVICE NO. CHANGE ****
15 REM
20 REM
100 INPUT"CURRENT DEVICE NUMBER=";CN
105 INPUT"REQUIRED DEVICE NUMBER=";RN
110 A=RN+32:B=RN+64
115 OPEN15,CN,15
120 PRINT#15,"M-W"CHR$(119)CHR$(0)CHR$(2
)CHR$(A)CHR$(B)
130 CLOSE15
```

When the programme is used only the disk drive to be changed should be switched on unless each drive already has a different device number.

Once the device number has been changed and the second disk switched on whenever it is required to access that disk the new device number is used in the command:

SAVE"DEVICE CHANGE",9

would save the programme called "DEVICE CHANGE" to the disk in drive 9.

LOAD"ACCOUNTS",8 would load the programme "ACCOUNTS" from device 8 and so forth.

One of the big advantages in using several drives is the capability to use one disk for programmes and the second for data storage. You may for example have a long 'card index' programme that repeatedly loads data from sequential files. The space used by sequential files will obviously take away disk space for the programme. The conventional way around this is to use separate floppy disks for the programme and the data and insert

the relevant disk into the drive. By using two drives the data disk may be located in one drive and the programme disk in the other.

DISK ERRORS

Whenever the disk controller detects an error condition, for example by trying to LOAD a file that does not exist on the disk in use, it will report the type of error to the operator. When making use of disks inside a programme it is desirable to keep track of any error by means of checking the error status as held in the command channel.

```
10 REM ********************************
15 REM ** DISK ERROR SUBROUTINE **
20 REM ********************************
25 REM
30 REM
100 OPEN1,8,15
50000 INPUT#1,EN,M$,DT,DS
50005 IFEN<20THENRETURN
50010 SC=PEEK(53281)
50015 POKE53281,2
50020 PRINT"■";TAB(12)"DISK ERROR"
50025 PRINT"■■ERROR NUMBER■",EN
50030 PRINT"■■ERROR TYPE■",M$
50035 PRINT"■■DISK TRACK■",DT
50040 PRINT"■■DISK SECTOR■",DS
50045 PRINT"■■CONTINUE? <Y/N>"
50050 POKE198,0
50055 GETA$:IFA$=""THEN50055
50060 IFA$<>"N"THENPOKE53281,SC:PRINT"■":RETURN
50065 CLOSE1:END
```

The error channel is OPENed in line 100; this should be placed at the front of the programme and executed only once. The error channel should always be OPENed first and closed last to prevent any loss of data.

50000 INPUTs the error messages from the error channel and if the error number is less than 20 (no error present) it will return control to the main programme.

Lines 5005 to 50040 display the exact nature of the error on the screen and lines 50045 to 50065 allow you to end the programme or continue.

It is strongly recommended that you use this routine by GOSUB50000 whenever you use the disk drive from inside a programme. The device number is set to 8 but may be changed to suit multiple drive configurations if desired.

SCREEN STORAGE

The disk may be used to store graphic displays For example you may have a screen display that you wish to use in various parts of the programme or wish to set up several different screens in various places in a programme. The technique employed is similar to that used when storing a screen in RAM except that instead of the data being located in memory it is stored as a sequential file on disk.

The examples which follow are all designed to be used as subroutines and should be called with the GOSUBXXXXX command where XXXXX is the starting line number of the routine.

```
100 NA$="TEST"
50000 REM ** SAVE SCREEN TO DISK **
50005 OPEN2,9,2,"0:"+NA$+"S,W"
50010 FORN=0TO999
50015 A=PEEK(1024+N):PRINT#2,A
50020 A=PEEK(55296+N):PRINT#2,A
50025 NEXT
50030 CLOSE2
50035 RETURN
```

To "save" a screen first set the screen up to the desired format assign NA$ to the name you wish to call it and GOSUB50000.

To restore a previously "saved" screen set NA$ to the name of the screen to be loaded and GOSUB50000.

```
100 NA$="TEST"
50000 REM ** LOAD SCREEN FROM DISK **
50005 OPEN2,9,2,"0:"+NA$+"S,R"
50010 FORN=0TO999
50015 INPUT#2,A:POKE1024+N,A
50020 INPUT#2,A:POKE55296+N,A
50025 NEXT
50030 CLOSE2
50035 RETURN
```

DISK MAINTENANCE

Normally whenever it is necessary to carry out disk maintenance commands such as VALIDATE INITIALISE or COPY the commands are entered as a direct command string.

In order to assist in the management of disks a programme has been included that will enable the operation required to be chosen by a 'menu' and executed by the programme. The facilities offered include SCRATCH RENAME COPY NEW VALIDATE INITIALISE etc and will greatly assist in the management of disk maintenance.

```
100 POKE53281,12:POKE53280,12
110 PRINT"■"TAB(6)"■ DISK MANAGEMENT PROGRAMME"
120 GOSUB1050
130 PRINTTAB(8)"■■■ENTER OPERATION NUMBER"
140 PRINT"■■■■EW DISK"TAB(20)"1"
150 PRINT"■INITIALISE DISK"TAB(20)"2"
160 PRINT"■VALIDATE DISK"TAB(20)"3"
170 PRINT"■COPY FILE"TAB(20)"4"
180 PRINT"■RENAME FILE"TAB(20)"5"
190 PRINT"■SCRATCH FILE"TAB(20)"6"
200 INPUT"■■■";O$:IFVAL(O$)<1ORVAL(O$)>6THEN110
210 ONVAL(O$)GOTO220,360,440,520,670,820
220 PRINT"■"TAB(16)"■EW DISK"
230 PRINT"■■WARNING:THIS WILL ERASE ALL DATA ON DISK
240 INPUT"■■ENTER DISK NAME■ ";N$
250 IFLEN(N$)>13ORLEN(N$)<1THEN240
260 INPUT"■■ENTER DISK NUMBER■ ";DN$
270 IFLEN(DN$)>2ORLEN(DN$)<1THEN260
280 PRINT"■"TAB(16)"■EW DISK"
290 PRINT"■■■DISK CALLED ■";N$
300 PRINT"■■■DISK NUMBER ■";DN$
310 PRINT"■■■PLEASE WAIT WHILE DISK IS FORMATTED"
320 OPEN15,DN,15
330 PRINT#15,"N0:"+N$+","+DN$
340 CLOSE15:GOSUB930
350 GOTO1010
360 PRINT"■"TAB(13)"■INITIALISE DISK"
370 PRINT"■■PLEASE WAIT"
380 OPEN15,DN,15
390 PRINT#15,"I0:"
400 CLOSE15
410 GOSUB930
420 PRINT"■■■DISK IS NOW INITIALISED"
430 GOTO1010
440 PRINT"■"TAB(14)"■VALIDATE DISK"
450 PRINT"■■PLEASE WAIT"
460 OPEN15,DN,15
470 PRINT#15,"V0:"
480 CLOSE15
490 GOSUB930
500 PRINT"■■■DISK IS NOW VALIDATED"
510 GOTO1010
```

```
520 PRINT"□"TAB(16)"COPY FILE"
530 PRINT"■■■ENTER FILE NAME TO BE COPIED■ "
540 INPUT"■";NF$
550 IFLEN(NF$)<1ORLEN(NF$)>13THEN520
560 PRINT"■■■ENTER FILE NAME OF COPY■ ";NN$
570 INPUT"■";CN$
580 IFLEN(CN$)<1ORLEN(CN$)>13THEN560
590 IFCN$=NF$THEN520
600 PRINT"■■■PLEASE WAIT"
610 OPEN15,DN,15
620 PRINT#15,"C0:"+CN$+"=":+NF$
630 CLOSE15:GOSUB930
640 PRINT"■■■FILE CALLED ■"TAB(20)NF$
650 PRINT"■■IS NOW COPIED AS■ "TAB(20)CN$
660 GOTO1010
670 PRINT"□"TAB(16)"■RENAME FILE■■"
680 PRINT"■ENTER OLD FILE NAME■"
690 INPUTNA$
700 IFLEN(NA$)<1ORLEN(NA$)>13THEN670
710 PRINT"■■ENTER NEW FILE NAME■"
720 INPUTNB$
730 IFLEN(NB$)<1ORLEN(NB$)>13THEN710
740 IFNA$=NB$THEN670
750 PRINT"■■■PLEASE WAIT"
760 OPEN15,DN,15
770 PRINT#15,"R0:"+NB$+"=":+NA$
780 CLOSE15:GOSUB930
790 PRINT"■■■FILE CALLED ■"TAB(20)NA$
800 PRINT"■■IS NOW ■ "TAB(20)NB$
810 GOTO1010
820 PRINT"□"TAB(14)"■SCRATCH FILE"
830 PRINT"■■■WARNING: THIS WILL ERASE THE FILE"
840 PRINT"■■ENTER FILE NAME■"
850 INPUTNA$
860 IFLEN(NA$)<1ORLEN(NA$)>13THEN820
870 PRINT"■■■PLEASE WAIT"
880 OPEN15,DN,15
890 PRINT#15,"S0:"+NA$
900 CLOSE15:GOSUB930
910 PRINT"■"NA$;"■ IS NOW SCRATCHED"
920 GOTO1010
930 INPUT#1,A,B$,C,D
```

```
940 IFA=0THENRETURN
950 PRINT"■"TAB(15)"■DISK ERROR"
960 PRINT"■■■ERROR NO.■"TAB(15)A
970 PRINT"■■■■DESCRIPTION■"TAB(15)B$
980 PRINT"■■■■TRACK NO.■"TAB(15)C
990 PRINT"■■■■SECTOR NO.■"TAB(15)D
1000 RETURN
1010 INPUT"■■■■CONTINUE ? <Y/N>";C$
1020 IFLEFT$(C$,1)="Y"THENCLOSE1:RUN
1030 IFLEFT$(C$,1)="N"THENCLOSE1:SYS64738
1040 GOTO1010
1050 PRINT"■■PLEASE ENTER DEVICE NUMBER"
1060 INPUTDN
1070 OPEN1,DN,15
1080 RETURN
```

USING A PRINTER

In addition to the obvious use of a printer to obtain listings of programmes a printer may also be used to generate graphics diagrams address labels and hard copy of the screen.

The programme that follows will print name and address on self-adhesive labels $1\frac{1}{2}$ by $3\frac{1}{2}$ inches in size on perforated backing paper with 4 inches between the holes.

```
10 REM ********************************
15 REM *** NAME AND ADDRESS LABELS ***
20 REM ********************************
25 REM
30 REM
100 POKE53281,15:PRINT"■"
105 INPUT"■■NAME■";N$
110 FORN=1TO4
115 INPUT"■■ADDRESS■";A$(N)
120 NEXT
125 INPUT"■■POST CODE■";PC$
130 PRINT"■■PLEASE CHECK LABEL POSITION"
135 PRINT"■■PRESS P TO PRINT OR S TO START AGAIN"
140 POKE198,0
145 GETA$:IFA$=""THEN145
150 IFA$="P"THEN165
155 IFA$="S"THENRUN
160 GOTO140
165 OPEN4,4
170 PRINT#4,N$
175 FORN=1TO4:PRINT#4,A$(N):NEXT
180 PRINT#4,PC$
185 FORN=1TO2:PRINT#4:NEXT
190 PRINT#4:CLOSE4
195 GOTO135
```

The programme will allow one line for the name and up to four lines for the address the first of which may be the company name. If RETURN is pressed without any text being entered that line will be left blank. The final line is for a post code.

Once the data has been entered you will have the opportunity to correctly position the label prior to printing taking place.

After the printing has finished you may print another copy of the label or enter new information. An example of a label is printed below.

```
MICROBOOKS
443 MILLBROOK ROAD
SOUTHAMPTON
HAMPSHIRE

SO1 0HX
```

CONTROL CHARACTERS

When using a printer with the Commodore 64 especially the Seikosha GP100 VC the various modes are controlled by control characters as CHR$( ) strings. This is often confusing and difficulty may be experienced in remembering the required code.

The way to overcome this problem is to assign a variable whose name is similar to the function desired and use the variable in the PRINT# command.

```
100 LINE$=CHR$(10):  REM * LINE FEED
105 RTN$=CHR$(13):   REM * RETURN
110 GRAPHIC$=CHR$(8):REM * GRAPHIC MODE
115 DOUBLE$=CHR$(14):REM * DOUBLE WIDTH
120 STNDRD$=CHR$(15):REM * NORMAL MODE
125 SPC$=CHR$(16):   REM * POSITION
130 DOT$=CHR$(27):   REM * DOT POSITON
135 UPPER$=CHR$(145):REM * UPPER CASE
140 LOWER$=CHR$(17): REM * LOWER CASE
145 RVRSE$=CHR$(18): REM * REVERSE ON
150 ROFF$=CHR$(146): REM * REVERSE OFF
```

When it is necessary to send a control character simply replace the CHR$( ) with the appropriate variable.

SCREEN DUMP

It is often desirable to be able to produce 'hard copy' of the screen display.

The following programme will print the entire contents of the screen and checks for upper/ lower case or upper case/graphics to ensure that the copy is accurate.

The routine will NOT copy a bit mapped screen or any user defined characters that may be used. Sprites are 'transparent' to the routine and will not be displayed either.

```
50000 REM ** DUMP SCREEN TO PRINTER **
50005 V$=CHR$(146)
50010 IFPEEK(53272)=23THENGOSUB50080:GOT
O50025
50015 S$=CHR$(145)
50020 OPEN4,4:PRINT#4,S$
50025 SB=1024
50030 FORP=SBTO2023
50035 C=PEEK(P):C$=""
50040 IF(P-SB)/40=INT((P-SB)/40)THENPRIN
T#4,CHR$(13)+CHR$(15);
50045 IFC>128THENC=C-128:C$=CHR$(18)
50050 IFC<32ORC>95THENC=C+64:GOTO50060
50055 IFC>63ANDC<96THENC=C+128
50060 C$=C$+CHR$(C)
50065 IFLEN(C$)>1THENC$=C$+V$+S$
50070 PRINT#4,C$;:NEXT
50075 PRINT#4:CLOSE4:RETURN
50080 S$=CHR$(17)
50085 OPEN4,4,7:PRINT#4,S$
50090 RETURN
```

If you have any comments on this publication, or are considering writing a book yourself, please contact us at the address below:-

MICROBOOKS
443 Millbrook Road
Southampton
Hampshire
SO1 0HX

Books on the following computers are in the course of production:

BBC Machine Code
Dragon Machine Code
CBM 64 Machine Code

The programmes in this book are available from PHOTRONICS, together with pads to aid graphic programming.

Please supply, post free

_____ cassette(s) at £5.95 each
_____ disk(s) at £7.95 each
_____ CBM 64 Screen pad(s) at £3.95 each
_____ Sprite pad(s) at £3.95 each
_____ User–defined character pads at £3.95
_____ Free details on the above products.

NAME _____

ADDRESS _____

_____

_____

I enclose cheque/PO for £_____

DATE _____

Send to:-

PHOTRONICS
Dept. MB
59 Kennedy Road
Maybush
Southampton
SO1 6DR

THE SENSIBLE 64

This book provides the reader, whether experienced programmer or initiate to the world of computing, with a simple, concise explanation of the first sensible computer for business or leisure use.

Starting with the more fundamental aspects of this remarkable machine and progressing logically to complex aspects such as using a disk drive, graphic printer, multicolour and Sprite graphics, it covers those topics that are so often only found in manuals or immersed in technicalities.

Special attention has been given to ensure that the examples will run without errors and that upon finishing the book the reader will be able to tackle even seemingly complex tasks with ease and confidence.

TOPICS COVERED INCLUDE:

- THE KEYBOARD AND FUNCTION KEYS
- USER-DEFINED AND MULTICOLOUR GRAPHICS
- SCREEN DISPLAYS AND BIT MAPPING
- CREATING AND USING SPRITES
- SOUND EFFECTS AND MUSIC
- PROGRAMMING WITH A DISK DRIVE
- USING A GRAPHIC PRINTER

THE SENSIBLE 64 is a must for every Commodore 64 owner that wishes to use the machine to its maximum potential.

443 Millbrook Road
Southampton SO1 0HX

Micro books

£5.95 UK ONLY
ISBN 0 946705 00 3