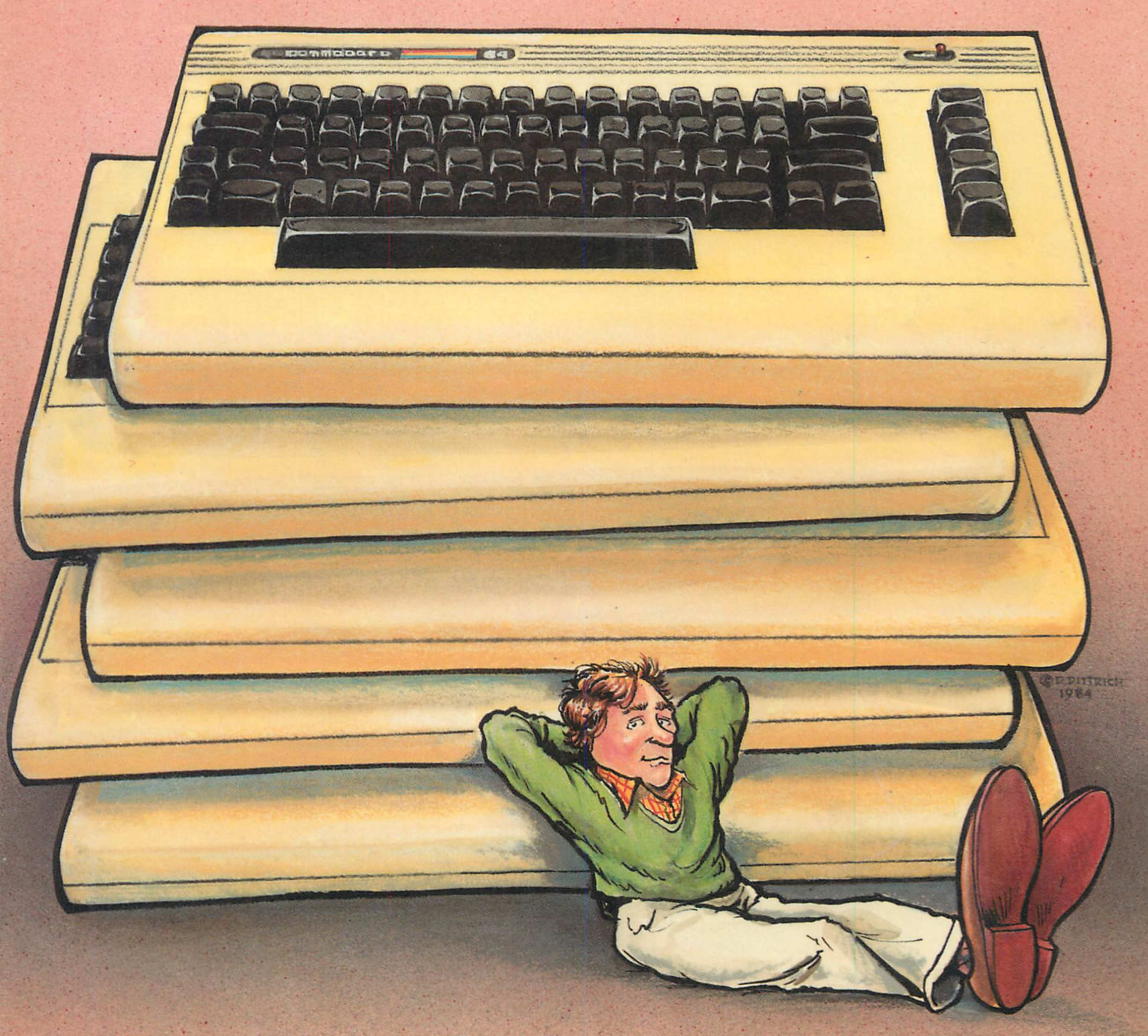
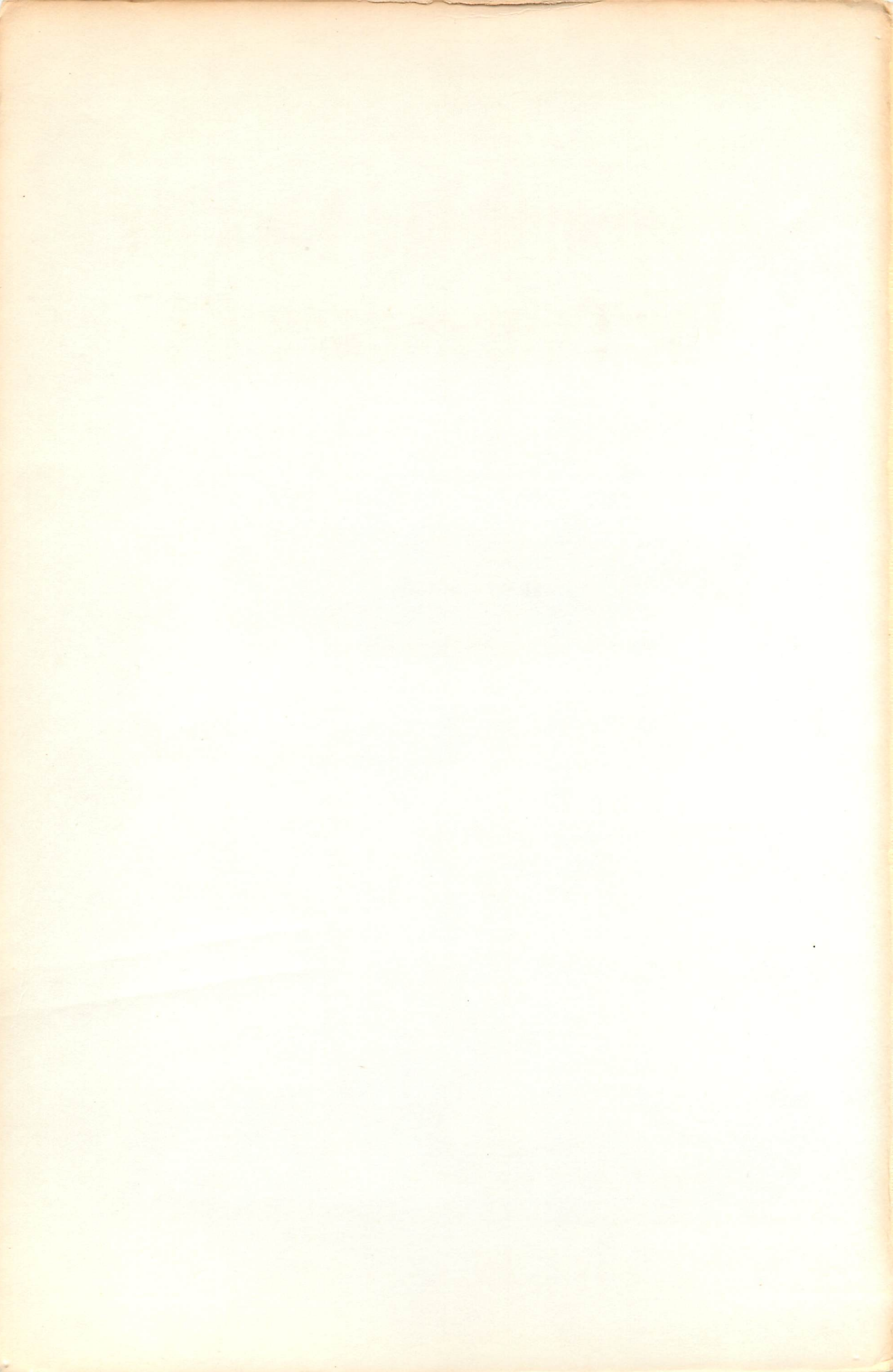
PROGRAMMING TIPS FOR THE COMMODORE 64™



DAVID HIGHMORE
LIZ PAGE



Programming Tips for the Commodore 64™

David Highmore
Liz Page

A Wiley Press Book
John Wiley & Sons, Inc.
New York • Chichester • Brisbane • Toronto • Singapore

Publisher: Judy V. Wilson
Editor: Theron Shreve
Managing Editor: Katherine Schowalter
Composition & Make-Up: The Publisher's Network, Morrisville, PA

Commodore 64™ is a registered trademark of Commodore Electronics, Ltd.

Copyright© 1985 by John Wiley & Sons, Inc.

All rights reserved. Published simultaneously in Canada.

Reproduction or translation of any part of this work beyond that permitted by Section 107 or 108 of the 1976 United States Copyright Act without the permission of the copyright owner is unlawful. Requests for permission or further information should be addressed to the Permissions Department, John Wiley & Sons, Inc.

Library of Congress Cataloging in Publication Data

Highmore, David.
Programming tips for the Commodore 64.
Includes index.
1. Commodore 64 (Computer—Programming. I. Page, Liz.
II. Title.
QA76.8.C64H54 1984 001.64'2 84-17395
ISBN 0-471-81553-5

Printed in the United States of America
84 85 10 9 8 7 6 5 4 3 2 1

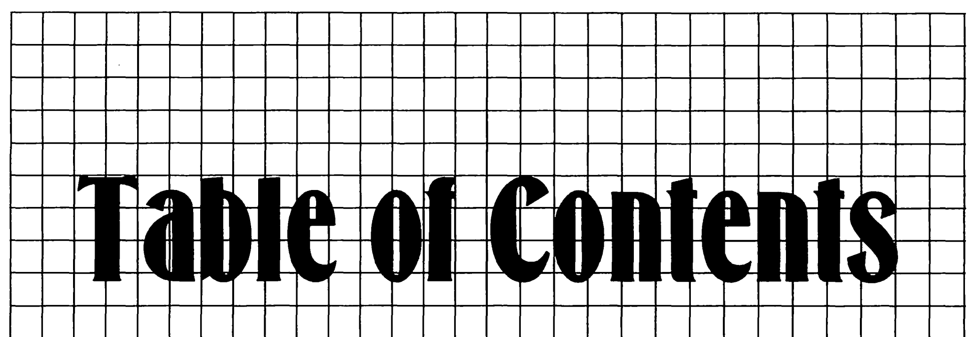


Table of Contents

Introduction vi

CHAPTER 1 Information Input 1

GET Statement 1

Function Keys 4

Input Control 8

CHAPTER 2 User-Defined Characters 11

Storing Characters 12

Reversed Characters 12

New Character Set 16

Multicolor Characters 18

CHAPTER 3 Sprites 21

Drawing a Sprite 22

Storing Sprite Data 23

Set Sprite Data Pointers 24

Sprite Characteristics 25

Enabling Sprites 28

Positioning Sprites 31

X MSB Register 33

Animation 38

Multicolor Sprites 40

Collision Detection 43

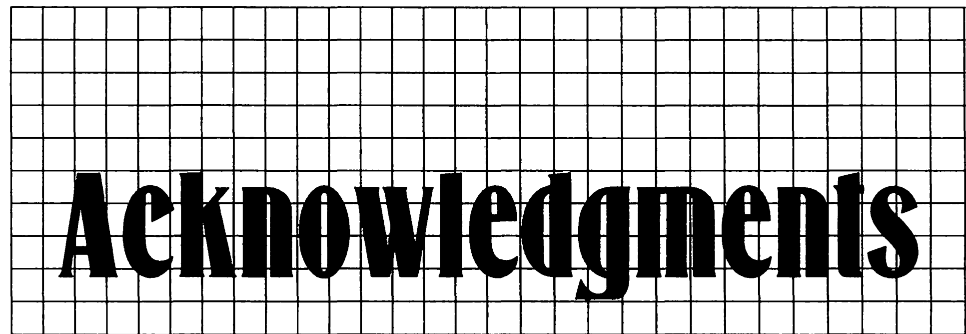
CHAPTER 4 The Screen **45**
Screen Format 45
Saving Screens 47
Machine Code Screen Saving 48
Screen Scrolling 50
Extended Color Mode 54
High Resolution Bit Mapping 56
X-Y Coordinate System 60
Shape Plotting 62
Multicolor Bit Mapped Graphics 66
Moving the Screen 70

CHAPTER 5 Joysticks **73**
Port Connections 73
Using Two Joysticks 76
Graphics 77

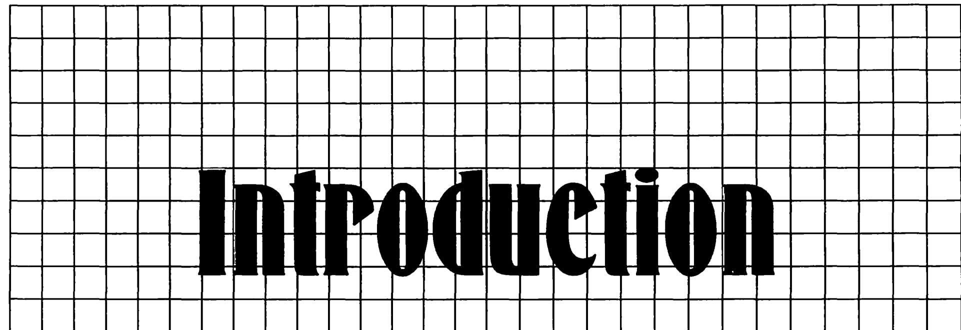
CHAPTER 6 Sound and Music **80**
Volume 81
ADSR 81
Pitch 84
Waveform 85
Sound Effects Programming 89
Music 91
Filtering 95
Resonance 97
Ring Modulation 97

CHAPTER 7 Disk Drives **99**
Device Numbers 99
Disk Errors 101
Screen Storage 102
Disk Maintenance 103

CHAPTER 8 Graphic Printer **104**
Name and Address Labels 104
Control Characters 105
Screen Dump 106

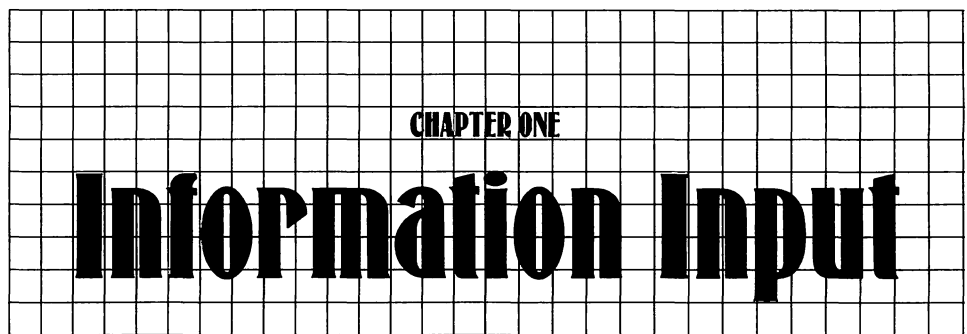


We'd like to thank Elizabeth Deal who helped us check all of our programs; also Katherine Schowalter who coordinated the production and Connie Kellner of The Publisher's Network who saw the project into print.



We wrote this book for anyone who has completed their Commodore 64 Manual and wants to go on to do more exciting things with their computer. We have made a special effort to provide help in areas which are difficult yet the most fun to explore.

For example, we show how to check keyboard input while interacting with your computer. We show how the computer plots bits for graphics and include numerous diagrams and short programs. This book provides helpful hints for better use of the computer's sound capabilities as is illustrated with a group of short programs.



There are times when you may wish to access information from the keyboard without using the INPUT command. For example, you may want to make use of the function keys so 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 (no key is pressed), it will return an empty string.

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

```
10 REM ** WAIT FOR ANY KEY **
15 PRINT"PRESS ANY KEY"
20 GOSUB10000
25 PRINT"OK THANK YOU"
30 END
10000 POKE 198,0
10005 GETA$:IFA$=""THEN10005
10010 RETURN
```

Line 10000 ensures there is no characters in the keyboard buffer. Alternative coding might be: GETA$,A$,A$.

Line 10005 waits until a key is pressed.

Line 10010 returns to the main program.

In this case, no information is passed back to the program and any key will cause the program 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
10015 IFA$<>"N"THEN10005
10020 Y=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 10030 returns to the main program.

There are two things that make this program an advance over the previous one: the computer will only accept a Y or an N; and the variable "flag" is set to indicate to the main program which key was pressed. This program can be adapted to select any two keys simply by substituting the keys for Y and N in lines 10010 and 10015.

If you wish to wait for a key in a particular range to be pressed, the ASCII code of the character can be compared to the value of the range required.

```
10 REM ** ONLY ALPHABET KEYS **
20 PRINT"PRESS ANY LETTER"
25 GOSUB10000
30 PRINT"OK. THANK YOU"
35 END
10000 POKE198,0
10005 GET A$:IF A$ = ""THEN 10005
10010 IFASC(A$)<65ORASC(A$))>90THEN10005
10015 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 returns to the main program.

The following program is essentially the same as the previous one except this one will wait until a number between 0 and 9 is pressed.

```
10 REM ** NUMBERS ONLY **
15 PRINT"ENTER A NUMBER 0 TO 9 "
20 GOSUB10000
25 PRINT"OK. THANK YOU"
30 END
10000 POKE198,0
10005 GET A$:IF A$ = ""THEN 10005
10010 IFASC(A$)<48 0R ASC(A$))>57THEN10005
10015 RETURN
```

The next program is useful in choosing an option from a menu: you can cause the program to branch, for example, to one of five parts of a program, based on which key is pressed.

```
10 REM ** CHOOSE LINE **
15 PRINT"ENTER NUMBER 1 TO 5"
20 GOSUB10000:ON A GOTO 100,200,300,400,500
100 PRINT"YOU CHOSE NUMBER 1":END
```

(continued)

```
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$)<49 OR ASC(A$)>53THEN10005
10015 A=VAL(A$)
10020 RETURN
```

Line 20 calls a subroutine, receives A, branches to any of lines 100 to 500 depending on A.

Line 10000 clears the keyboard buffer.

Line 10005 waits for a key to be pressed.

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

Line 10015 converts it into a numeric variable.

Line 10020 returns to the main program.

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

FUNCTION KEYS

All the previous examples 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, or PEEK(197). The following program will give the value of the key pressed.

```
10 REM ** PEEK FOR ANY KEY **
15 PRINT"PRESS ANY KEY"
20 GOSUB10000
25 PRINT"OK. THANK YOU"
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 program. While in the previous examples POKE198,0 was optional, here it is mandatory. It should be noted that if no key is pressed, PEEK(197) will be 64.

The next program we will look at is an advance of the previous program. This program is designed to wait until the SPACE BAR is pressed.

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

Line 10000 waits until PEEK(197) = 60 (when the space bar is pressed). Alternative coding: GETA$:IFA$""THEN10000, is machine independent.

Line 10005 clears the keyboard buffer.

Line 10010 returns to the program.

The following program demonstration will wait until the RETURN key is pressed. Once again, you can use GET: 10000 GET A$:IFASC(A$) < >13THEN10000, will do the job just as well.

```
10 REM ** ANY FUNCTION KEY **
15 PRINT"PRESS A FUNCTION KEY"
20 GOSUB10000
25 PRINT"OK. THANK YOU"
30 END
10000 N=PEEK(197)
10005 IFN<3 OR N>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 program.

The next program will wait until any of the FUNCTION KEYS are pressed.

```
10 REM ** CHECK A FUNCTION KEY***
15 PRINT"PRESS A FUNCTION KEY1(F-1)"
20 GOSUB10000
25 PRINT"OK. THANK YOU F-1 PRESSED"
30 END
10000 IFPEEK(1197)()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 POKE 198,0
10035 RETURN
```

Line 10000 waits until a specific function key is pressed from the value shown in 10002 to 10027. Alternative coding might be: 10000 GET A$:IFASC(A$) < >133 THEN 10000. ASC values of function keys 1,3,5,7 are 133 to 136.

Line 10030 clears the keyboard buffer.

Line 10035 returns to the main program.

The following program will enable you to use the function keys to make decisions.

```
10 REM ** CHOOSE FROM F-KEY **
15 PRINT"PRESS A FUNCTION KEY"
20 GOSUB10000:ON N GOTO 100,200,300,400
```

(continued)

```
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 1000
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 RETURN
```

Line 20 calls a subroutine, then branches depending on N.

Line 10000 sets variable N to PEEK(197).

Line 10005 checks to see if it is a function key.

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

Line 10030 clears the keyboard buffer.

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

Key numbers returned in location 197 ignore the SHIFT key. If you need to access all keys, GET *must* be used. The coding might be:

```
10 REM ** CHOOSE ANY FUNCTION KEY 1-8 **
20 REM ** ASCII VALUES ALTERNATE:
30 REM ** F1,F3,F5,F7  133,134,135,136
40 REM ** F2,F4,F6,F8  137,138,139,140
50 PRINT"PRESS ANY FUNCTION KEY,"
60 PRINT"SHIFTED OR UNSHIFTED>"
70 PRINT"FUNCTION-8 WILL QUIT.":PRINT
100 GOSUB 10000
110 PRINT"FUNCTION KEY" N "PRESSED":IF N<8 GOTO100
120 END
10000 GET A$:IFA$=""GOTO 10000
10010 N=ASC(A$)-132:IFN<1 or N>8 GOTO 10000
10015 N=2*N-1:IF N>7 THEN N=N-7
10020 RETURN
```

This section has dealt with a number of ways to control the way information input is handled. The next section shows you, through a range of techniques, how it is possible to control the input that is accepted.

INPUT CONTROL

The following program will enable you to input a word, one letter at a time, until the word is of the specified length.

```
10 REM ** ENTER WORD LENGTH **
50 INPUT"  WORD LENGTH";L
1000 W$="":PRINT"[CLR]"
1005 FORN=1 TOL<EP>
1010 PRINT"[HOME] ENTER WORD ";W$
1015 POKE198,0
1020 GETA$:IFA$=""THEN 1020
1025 W$=W$+A$
1030 NEXT
1035 PRINT"[HOME] 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.

By adding a few lines to the previous program it is possible to limit the input to only alphabet characters, only numbers, or even both. Line 1025 will check to make sure each character is a letter of the alphabet.

```
10 REM ** LIMIT TO ALPHABET **
50 INPUT"[CLR]WORD LENGTH";L
1000 W$="",PRINT "[CLR]"
1005 FORN=1TOL
1010 PRINT"[HOME]ENTER WORD":W$
1015 POKE 198,0
1020 GET$A:IFA$=""THEN 1020
1025 IFASC(A$)(650RASC(A$)) 90THEN 1015
1030 W$=W$+A$
1040 PRINT"[HOME]WORD ENTERED IS ";W$
```

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

```
10 REM ** LIMIT TO NUMBERS **
50 INPUT"[CLR] NUMBER LENGTH";L
1000 W$="":PRINT"[CLR]"
1005 FORN=1TOL
1010 PRINT"[HOME] ENTER NUMBER ";W$
1015 POKE198,0
1020 GETA$:IFA$=""THEN 1020
1025 IFASC(A$)<48 0R ASC(A$))57THEN 1015
1030 W$=W$+A$
1035 NEXT
1040 N=VAL(W$)
1045 PRINT"[HOME] NUMBER ENTERED IS ";W$
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 it does when using the INPUT statement. This will allow you to ensure that only required characters will be accepted.

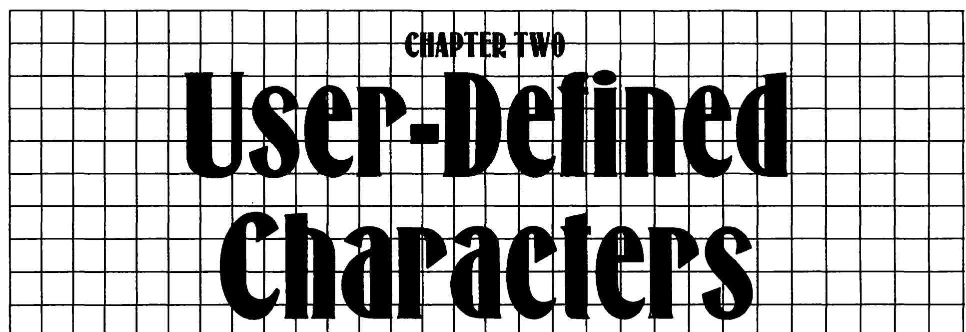
```
10 REM ** USE RETURN TO TERMINATE **
1000 W$="":PRINT"[CLR]"
1005 PRINT"ENTER WORD ";W$
1010 POKE198,0
1015 GETA$:IFA$=""THEN 1015
1020 IFA$=CHR$(13)THENA$="":GOTO1045
1025 IFASC(A$)<65 0R ASC(A$)>90THEN1010
1030 W$=W$=A$
1035 PRINT"[HOME] ENTER WORD ";W$
1040 GOTO1005
1045 PRINT"[HOME] THE WORD 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.

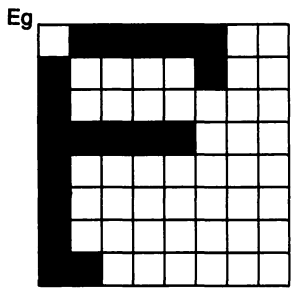
Of course, like the last few programs, this one may be modified to accept numbers or any other characters.

It is also possible to combine this small routine with others to perform a very powerful set of functions. The range of applications is wide—from games to business programs.



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

Characters on the Commodore 64 are defined on an 8 by 8 matrix or grid. The grid shown below defines the letter "F".



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; every blank requires a 0. The eight rows of the "F" are represented by the following eight bytes.

```
Row 1 01111100
Row 2 10000100
Row 3 10000000
Row 4 11111000
Row 5 10000000
Row 6 10000000
Row 7 10000000
Row 8 11000000
```

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

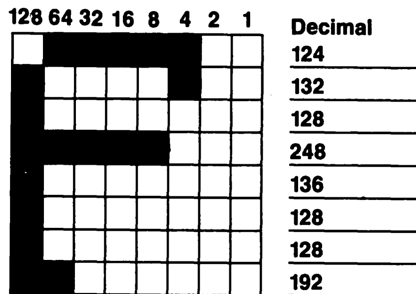
STORING CHARACTERS

The character ROM in the Commodore 64 contains the information to form all the characters available. Since this information is in Read Only Memory, you may wonder how it is possible to define your own characters. This is achieved by directing the VIC chip (which controls the screen display) to look for the information in a different place. Because you want to store your own data, this place must be in RAM. The only RAM readily available is also used by Basic so 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 programs 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 POKE51,0:POKE52,48
60 POKE55,0:POKE56,48
70 POKE53272,(PEEK(53272)AND240)+12
```

When you run this program, 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 described by an 8 bit binary number. All that has to be done now is to convert that number into its decimal equivalent. This is done by adding up the binary place-value equivalent of each filled square (see example on the next page).




```
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 POKE51,0:POKE52,48
120 POKE55,0:POKE56,48
130 FORN=0TO7
140 READA
150 POKE12288+N,A
160 NEXT
170 PRINT"[CLR 2DOWN 2RIGHT @]"
180 DATA124, 132, 128,248, 136, 128, 128, 192
```

This program alters the area from which the character data is read, lowers the top of memory to protect the data from being overwritten, and stores the data for the new character in the space normally used to hold the @ sign. As many characters as you wish may be defined and used in this way. The data starts at 12288 and is held in 8 bytes. Subsequent characters will be stored in 8 byte increments, for example:

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

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 POKE51,0:POKE52,48:POKE55,0:POKE56,48
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 56334 & RESTORES
CHAR ROM
```

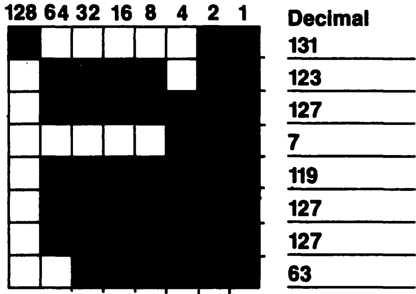
This program will copy the character ROM into RAM starting at 12288. However, because the Commodore 64 has a 64K RAM, the character ROM space is normally used for different purposes and is not immediately available. This program disables the key scan and enables the character ROM, PEEKs the data byte by byte and POKES it into the RAM, then restores the normal key scan function 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, (@ = 0, A = 1, Z = 26, etc.).

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

REVERSED CHARACTERS

To create a reversed character, a 1 is substituted for every 0 and a 0 for 1. Going back to the example of the F, this program uses the same data but reverses the bit pattern of each row to form a reversed character.




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

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

```
Row 1 = 255 - 124 = 131
Row 2 = 255 - 132 = 123
Row 3 = 255 - 128 = 127
```

(continued)

```
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 program demonstrates a character reversal.

```
10 REM ** REVERSED 'F' **
100 POKE53272,28
110 POKE55,0:POKE56,48
120 POKE51,0:POKE52,48
130 FORN=0t07
140 READA
150 POKE12288+N,255-A
160 NEXT
170 PRINT"[CLR 2DOWN 2RIGHT @]"
180 DATA124,132,128,248,136,128,128,192
```

NEW CHARACTER SET

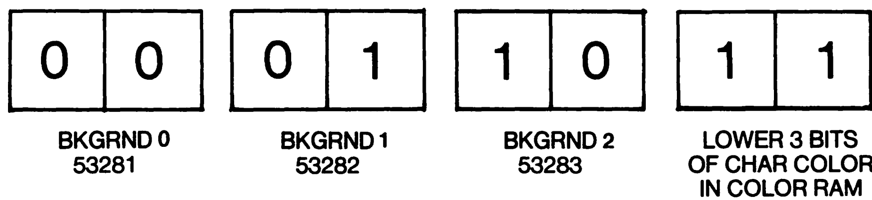
The following program will replace the Commodore 64's normal character set with "space age" style font.

```
10 REM ** NEW CHARACTER SET<EP>
100 FOR N=0 TO 79<EP>
105 READ A<EP>
110 POKE 12288+384+N,A<EP>
115 NEXT<EP>
120 REM<EP>
125 REM READ NUMERAL DATA & STORE IN RAM<EP>
130 REM<EP>
135 REM<EP>
200 FOR N=0TO207<EP>
205 READ A:POKE12288+8+N,A<EP>
210 NEXT<EP>
```

```
215 REM READ ALPHABET DATA & STORE IN RAM<EP>
220 REM<EP>
225 REM<EP>
230 POKE53272,28<EP>
235 REM<EP>
240 REM<EP>
245 REM SWITCH TO NEW CHARACTER SET<EP>
1000 DATA126,70,74,82,102,70,126,0:   REM 0<EP>
1001 DATA8,8,8,24,24,24,0:REM 0:      REM 1<EP>
1002 DATA126,2,2,126,96,96,126,0:     REM 2<EP>
1003 DATA126,2,2,62,6,6,126,0:        REM 3<EP>
1004 DATA64,68,68,126,12,12,12,0:     REM 4<EP>
1005 DATA126,64,64,126,6,6,126,0:     REM 5<EP>
1006 DATA64,64,64,126,98,98,126,0:    REM 6<EP>
1007 DATA126,66,66,6,6,6,6,0:         REM 7<EP>
1008 DATA126,66,66,126,98,98,126,0:   REM 8<EP>
1009 DATA126,70,70,126,2,2, 2 ,0:     REM 9<EP>
2000 DATA126,66,66,126,98,98,98,0:    REM A<EP>
2001 DATA124,68,68,126,98,98,126,0:   REM B<EP>
2002 DATA126,66,66,96,98,98,126,0:    REM C<EP>
2003 DATA126,66,66,98,98,98,126,0:    REM D<EP>
2004 DATA126,64,64,124,96,96,126,0:   REM E<EP>
2005 DATA126,64,64,124,96,96,96,0:    REM F<EP>
2006 DATA126,66,64,102,98,98,126,0:   REM G<EP>
2007 DATA66,66,126,98,98,98,0:        REM H<EP>
2008 DATA16,16,24,24,24,24,0:         REM I<EP>
2009 DATA2,2,2,6,70,70,126,0:         REM J<EP>
2010 DATA66,66,66,124,98,98,98,0:     REM K<EP>
2011 DATA64,64,64,96,96,96,126,0:     REM L<EP>
2012 DATA126,74,74,106,106,106,106,0:REM M<EP>
2013 DATA98,82,74,70,98,98,98,0:      REM N<EP>
2014 DATA126,66,66,70,70,70,126,0:    REM O<EP>
2015 DATA126,66,66,126,96,96,96,0:    REM P<EP>
2016 DATA126,66,66,106,106,126,8,0:   REM Q<EP>
2017 DATA124,68,68,126,98,98,98,0:    REM R<EP>
2018 DATA126,98,96,126,2,66,126,0:    REM S<EP>
2019 DATA126,16,16,24,24,24,24,0:     REM T<EP>
2020 DATA66,66,66,98,98,98,126,0:     REM U<EP>
2021 DATA66,66,66,102,36,24,24,0:     REM V<EP>
2022 DATA74,74,74,106,106,126,126,0:  REM W<EP>
2023 DATA66,66,126,24,126,66,66,0:    REM X<EP>
2024 DATA66,66,66,126,24,24,24,0:     REM Y<EP>
2025 DATA126,66,68,8,18,34,126,0:     REM Z<EP>
```

MULTICOLOR CHARACTERS

In the preceding examples, the user-defined characters have been in only two colors—the background color 0 and the character color. This is because each bit controls one dot (pixel) on the screen and each bit can contain a 0 or a 1. With only two possible combinations there can be only two colors. 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 colors as a pair of bits may have one of four values.



The color Ram starts at 52296. Colors 0 to 7 are normal and 8 to 15 are in multicolor mode, which can be set by the PRINT command.

POKE 53270,PEEK(53270) OR 16 will enable multicolor.
POKE 53270,PEEK(53270)AND239 will disable multicolor.

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

The first color caused when two adjacent bits are set to 0 is the screen color set in location 53281. The second color is produced when the bit pattern is 01 and is held in location 53282. The third color is produced by a 10 pattern and is stored at location 53283. The fourth color is the character color 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 (high resolution). If it is between 8 and 15, the character is printed multicolor—the fourth color dependent on the actual value. This enables both high resolution and multicolor graphics to be used on the same screen. However only colors 0 to 7 (black to yellow) may be utilized for high resolution colors.

It is important to remember when designing multicolor characters that although each bit is treated individually for calculation purposes, it is the

combination of two adjacent bits that determines the color, and that the pixel size is effectively doubled. Thus the character matrix is four columns by eight rows when the multicolor mode is used.

The diagram below shows the correspondence between the bit pattern and color 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

128 64 32 16 8 4 2 1

"Bkgnd Screen Color"
Bkgnd 1
Bkgnd 2
Char color

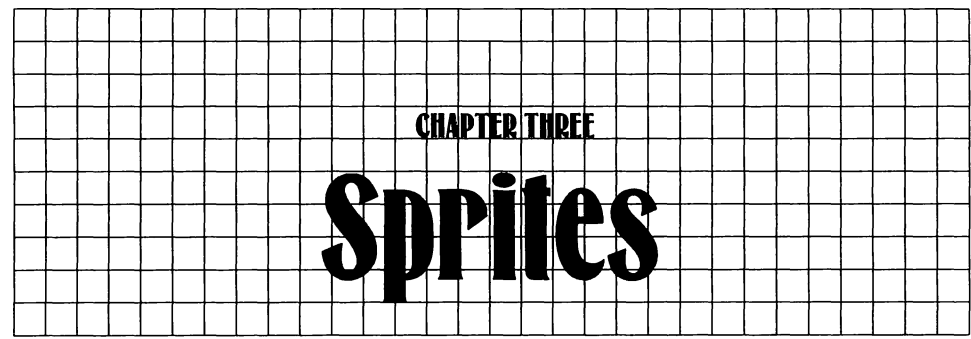
In this example, the colors are set at:

Screen or Background—light gray (15)
Background 1—green (5)
Background 2—magenta (4)

```
10 REM ** USER DEF MULTICOLOR **
100 PRINT "[CLR]" :POKE53272,28
105 POKE55,0:POKE56,48
110 POKE51,0:POKE52,48
115 POKE53270,PEEK(53270)OR16
120 POKE53281,15:POKE53282,5
125 POKE53283,4
130 FORN=0T07
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
```

The first half of the screen RAM is loaded with 15, which enables multicolor in that square and sets the fourth color 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. This is why only colors 0 to 7 may be employed for normal user-defined characters in multicolor mode.



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 multicolor mode.

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

The steps necessary to define and use sprites can be summarized as follows:

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. color
   b. normal or multicolor
   c. background priority
   d. horizontal or vertical expansion
5. Enable the sprite.
6. Position the sprite.

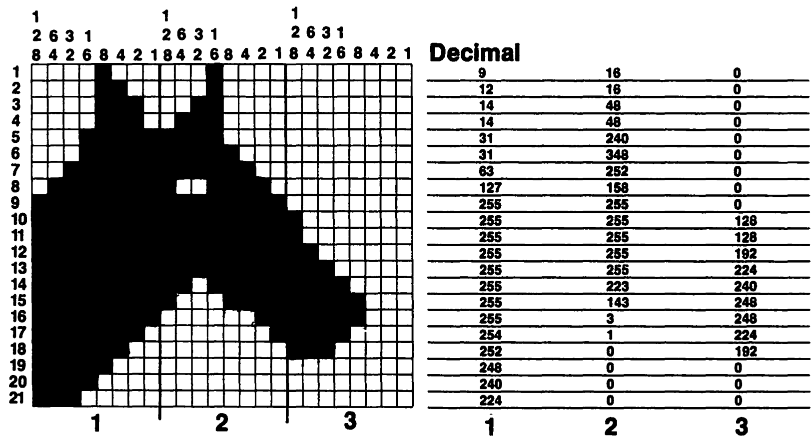
Since each sprite is controlled independently of all other graphics, including other sprites, it is possible to have many sprites, each with differing characteristics.

In order to explain clearly 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.

DRAWING A SPRITE

Sprites are formed in a way analagous to user-defined characters except that the grid used consists of 21 rows and 24 columns.

The appearance of a sprite drawn on the grid is shown below.



Decimal

Row	1	2	3
1	9	16	0
2	12	16	0
3	14	48	0
4	14	48	0
5	31	240	0
6	31	348	0
7	63	252	0
8	127	158	0
9	255	255	0
10	255	255	128
11	255	255	128
12	255	255	192
13	255	255	224
14	255	223	240
15	255	143	248
16	255	3	248
17	254	1	224
18	252	0	192
19	248	0	0
20	240	0	0
21	224	0	0

The rows are divided into three sets of eight squares.

Each eight-square section is 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, and so on, 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

The rules for determining the value for each 8 bit row are identical to those used for user-defined characters and, if you need to review this process, read the section on user-defined characters.

STORING SPRITE DATA

Once the sprite has been drawn and the decimal conversion is complete, the next stage is to store the information in RAM.

First you must decide where to put it. Each sprite will require an effective total of 64 bytes of memory. Usually this is within the first 16K of RAM as the VIC II chip that controls sprites can only "see" up to location 16383.* This area is also used by Basic and will limit the space for programs to about 10K.

If you do not wish to define more than three different sprites there is a solution: 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 transferred to and from the cassette unit. If the tape unit is used, the sprite data will be lost, but 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 program so there is little conflict between the two uses.

The following program uses the horse's head example to illustrate the simplest way to store the data.

*This assumes a normal C-64 setup. Techniques exist for 'banking' where the VIC Chip can look at other memory. This is beyond the scope of this book, but a few details are provided.

```
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, read into variable A in the FOR-NEXT loop, and stored in RAM starting at location 832. The reason for this data location will become apparent when we examine the procedure for location of the sprite data.

If you were to use more than three 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 found in the section on user-defined characters.

SET SPRITE DATA POINTERS

The next step is to inform the computer of the location of the data for each sprite.

In the normally used Bank 0, 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 as POKE2040,13. If you had the data loaded at 12288, the POKE would be 12288/64 = 192 and be set as POKE2040,192.

The same set of data may be utilized 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 four 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, which cannot be divided by 64 without a remainder.

SPRITE CHARACTERISTICS

Color

The color of each sprite is determined by the contents of the SPRITE COLOR REGISTER, located as follows:

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

Any of the 16 normally available colors may be utilized. The code for the color required is POKEd into the relevant color register; for example, POKE53287,0. All the bits set to 1 in sprite 0 will appear in black. The bits set to 0 will be "transparent" and will show the color of whatever the sprite is positioned over. Normally this is the screen color (held in 53281).

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

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 register is located at 53275. Each bit 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 as follows:

```
POKE53275,PEEK(53275)OR2^N <EP>
```

This will cause the sprite to be displayed behind other characters where N is the number (0 to 7) of the sprite. The following will set the sprite to a higher priority:

```
POKE53275,PEEK(53275)AND(255-2^N) <EP>
```

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	v
7	Lowest

Sprite Expansion

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

BIT	SPRITE	EXPANSION REGISTER
0	0	0 = Normal size
1	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:

```
POKE 353271, (PEEK(53271)OR(2^N)
```

To return to normal:

```
POKE 53271,PEEK (53271)AND(255-2^N)
```

where N is the number of the sprite.

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

ENABLING SPRITES

Now that the characteristics of the sprites have been established you are ready to enable the sprite. Each sprite may be enabled (turned on) or disabled at will. This is controlled by the SPRITE ENABLE REGISTER, located at 53269. As 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 is on; 0 is off
1	1	1 is on; 0 is off
2	2	1 is on; 0 is off
3	3	1 is on; 0 is off
4	4	1 is on; 0 is off
5	5	1 is on; 0 is off
6	6	1 is on; 0 is off
7	7	1 is on; 0 is off

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

To enable sprite N:

```
POKE 53269,PEEK(53269)OR(2^N)
```

or to disable a sprite—

```
POKE53269, PEEK*53269)AND(255-2^N)
```

Where N is the number of the sprite in question.

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

The complete program up to this point is:

```
50 POKE53281,11
55 POKE53280,12
60 REM **SET SCREEN & BORDER COLOR**
65 REM ------------------------------------
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 COLOR TO WHITE **
155 REM -----------------------------------
160 POKE53275,PEEK(53275)OR2^0
165 REM ** SPRITE WILL APPEAR BEHIND **
170 REM -----------------------------------
175 POKE53271,PEEK(53271)AND(255-2^0)
180 REM ** NO VERTICAL EXPANSION **
185 REM -----------------------------------
190 POKE53277,PEEK(53277)AND(255-2^0)
195 REM ** NO HORIZONAL EXPANSION **
200 REM -----------------------------------
205 POKE53269,PEEK(53269)OR2^0
210 REM ** TURN ON SPRITE **
215 REM -----------------------------------
10000 DATA8,16,0
10005 DATA12,16,0
10010 DATA14,48,0
10015 DATA14,48,0
```

(continued)

```
10020 DTAT31,240,0
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:

Line 50 set screen color to gray

Line 55 set border color to dark gray

Line 100 start counting loop

Line 105 read sprite data from data statements

Line 110 load data into RAM starting at 832

Line 115 increment loop count

Line 130 set data pointer to 832 (832/64 = 13) for sprite 0

Line 145 set sprite 0 color to white

Line 160 ensure sprite 0 priority is low

Line 175 ensure sprite will appear normal size in the y direction (vertical)

Line 190 ensure sprite will appear normal size in the x direction (horizontal)

Line 205 enable sprite 0

Line 10000 to **Line 10100** contain the sprite data arranged in columns and rows (compare with the diagram of the horse's head sprite).

With all this done, running the program will only cause the screen and the border to change color. This is because the sprite has not yet been positioned on the screen.

POSITIONING SPRITES

For the purposes of sprite positioning, the screen may be viewed as consisting of 255 vertical and 344 horizontal positions. Each sprite is 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 true regardless of whether or not the sprite has been expanded.

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

In the horizontal direction, the limit is set at 344—the position at which 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 50-299 vertically and 24-320 horizontally.

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

LOCATION	SPRITE	POSITION	
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

Starting position can be thought of as address VI = 53248. If sprite number is N, then its X position is at VI + 2*N, and its Y position is at VI + 2*N + 1.

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 onto the screen. POKE53248,100:POKE54349,100 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, demonstrating the ability to animate displays. Add the lines below to the program 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=1TO5-:NEXTT
240 NEXTN
245 REM * MOVE HORSE FROM TOP TO BOTTOM *
250 REM
```

Line 220 sets the horizontal position.

Line 225 starts the counting loop.

Line 230 POKEs the vertical position into the Y register.

Line 235 slows down the movement.

Line 240 increments the loop.

To move the sprite back to the top of the screen, simply change line 255:

```
225 FORN=229TO50STEP-1
```

The horse will now move from the bottom to the top of the screen.

By altering the program 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)OR2^0
210 REM **  TURN ON SPRITE        0**
215 REM
220 POKE53249,100
225 FORN=24 TO255
230 POKE53248,N
235 FORT=1TO50:NEXTT
240 NEXTN
245 REM * MOVE HORSE FROM LEFT TO RIGHT *
250 REM
```

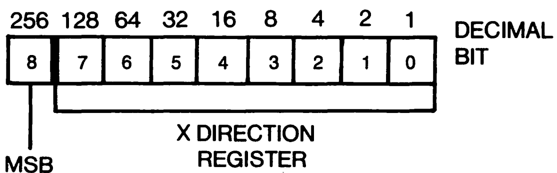
You will now see the horse move from left to right. However, the motion stops about three-quarters of the way across. You will recall that the screen is 24 to 320 "steps" across. Because the position register is one byte, and one byte on the Commodore 64 is 8 bits wide, the maximum decimal value that 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.

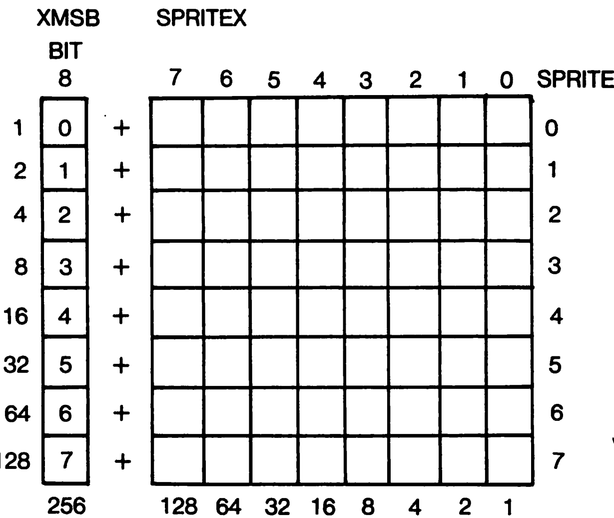
X MSB Register

The X direction MSB (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 byte is 256 + 128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = 511, you can position your sprite wherever you wish.

Each bit in the MSB register is effectively the highest (8th) bit for each of the 8 sprite X direction registers and may be viewed like this:



Whenever you want to position a sprite beyond 255, you must set the MSB bit corresponding to that sprite, 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. This 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	1	256
1	1	257
1 64	320	320 (maximum value)

Setting the MSB

You must make sure 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:POKEX 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(255-2^N):POKE XREG,ANY
 X VALUE
```

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

You can now modify the program 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 **
```

Line 255 set MSB for sprite 0.

Line 270 clear normal X register.

Line 280 start loop.

Line 285 load register with position.

Line 290 slow down movement.

Line 295 increment loop.

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

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

Line 220 clears the screen and starts counting loop.

Line 225 defines X coordinate of circle.

Line 230 defines Y coordinate of circle.

Line 235 POKEs X coordinate into register.

Line 240 POKEs Y coordinate into register.

Line 245 increments loop.

Up to this point we were thinking of the X position as divided into two parts. The part from 0 to 255 and the part from 255 up. In the first case the MSB bit was cleared. In the second case it was set.

If you can afford a bit of time penalty for additional calculations, the following shows a more general way. Assume that sprite number is N, VIC chip registers are at VI = 53248, X is the new horizontal position in the range 0-320, or whatever values you need:

```
X%=X/256
POKE VI+16, (PEEK(VI+16)) OR X%*(2^N)
POKE VI+1*N,X-256*X%
```

This is a bit slower, since it has to look at the MSB register in all instances. But if time is of not much importance, it is a feasible way of coding two jobs in one.

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
```

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

```
50 POKE53281,11
55 POKE53280,12
60 REM ** SET SCREEN & BORDER COLOR **
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 COLOR 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 "[CLR]"
230 FORN=0TO8:PRINT:NEXT
235 FORN=0TO199:PRINT"[BLACK LOGO-I]";:NEXT
```

(continued)

```
240 FORN=0TO255
245 POKE53249,N
250 FORT=1TO50:NEXT
255 NEXT
260 END
```

If line 160 is changed to:

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

the sprite will move over the barrier.

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 following program:

```
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)OR2^0
115 POKE53248,100:POKE53249,100
120 POKE2040,13
125 FORT=1TO500:NEXTT
130 POKE2040,14
135 FORT=1TO500:NEXTT
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 DATA 0,60,12
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 DATA0,225,0
10120 DATA0,225,0
```

Line 10000-10020 hold the data for the 1st shape

Line 10100-10120 hold the data for the 2nd shape

Line 100 reads and stores the data for the 1st shape

Line 105 reads and stores the data for the 2nd shape

Line 110 turns on sprite 0

Line 115 positions the sprite on the screen

Line 120 sets the data pointer to the 1st shape

Line 125 delays the animation

Line 130 sets the data pointer to the 2nd shape

Line 135 delays the animation

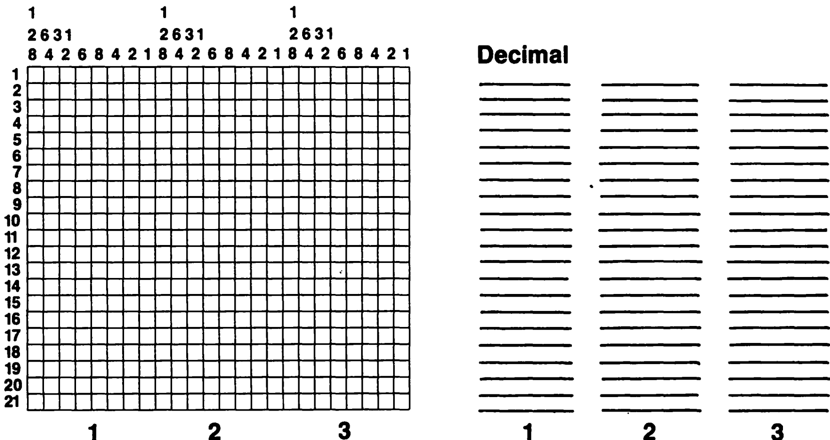
Line 140 returns to continue the effect

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 you use this method until you are familiar with the relationship.

Once the program has been written and the sprite data is known to be correct, the data statements can be "run together" to conserve memory space.

MULTICOLOR 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 multicolor mode. This will allow up to four colors to be used in each sprite. As usual, when two bits are required to select the color, the horizontal resolution is cut in half.



As you can see from the above diagram, the sprite now consists of 12 horizontal and 21 vertical sections. The bit pattern in each section controls the color of that area. The relationship is:

BIT PATTERN	COLOR DISPLAYED
00	Transparent (screen)
01	Sprite multicolor 1 (53285)
10	Sprite normal color
11	Sprite multicolor 2 (53286)

It will be apparent that three colors are common to each sprite, and one color may be selected independently.

The SPRITE MULTICOLOR ENABLE REGISTER controls each sprite and any, all or no sprites may be displayed in multicolor mode. The register is located at 53276 and, once again, each bit controls one sprite; bit 0 controlling sprite 0, bit 1 controlling sprite 1, and so on.

To set a sprite to multicolor, the corresponding bit in the register must be set to a 1. To display a sprite in normal mode, the corresponding bit is set to a 0.

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

```
POKE53276,PEEK(53276)OR2^N
```

where N is the number of the sprite to be changed into multicolor mode.

```
POKE53276,PEEK(53276)AND(255-2^N)
```

will restore a multicolor sprite to normal mode.

The colors are selected by the contents of four registers.

Bits 10 will be set in the normal sprite color register, thus permitting each sprite to have a different color 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 MULTICOLOR 2, held in 53286. The codes are the same for all sprites and may contain any color code from 0 to 15.

The following program will demonstrate what a multicolor sprite may look like.

```
10 REM **********************************
15 REM ** MULTICOLOR 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-COLOR 1  01  **
120 POKE53286,6: REM * M-COLOR 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 a sprite-sprite or sprite-display collision on the screen. The collision status for each sprite is held as a bit in two registers called the SPRITE COLLISION REGISTERS. One register holds sprite-to-sprite collision status; the other, sprite-to-background status.

Their location is:

```
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 following formula is used:

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

This checks to see if sprite N has had a collision: if it has, then do XXXXXX (XXXXXX is whatever action is required).

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

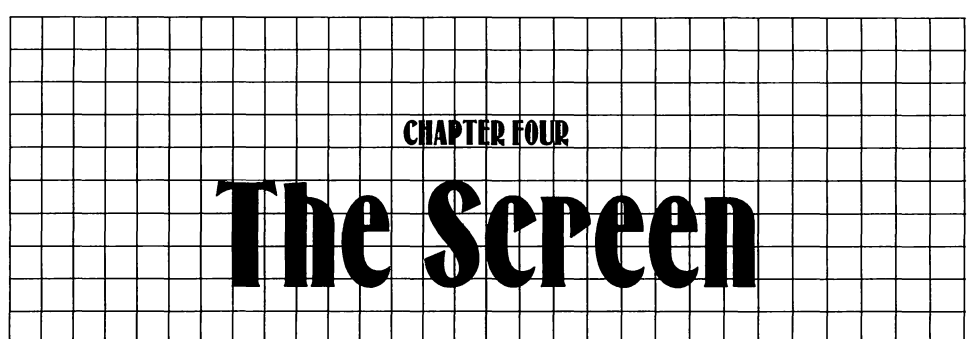
Notes on Collisions

The collision status will be set even if the sprite is off the screen at the time of impact, this is relevant only for the sprite-sprite crashes.

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 multicolor mode, the bit pattern 01 (multicolor 1), although visible, is considered to be transparent as far as collisions are concerned: you must ensure that the program will not attempt to detect a collision with this pattern.



CHAPTER FOUR

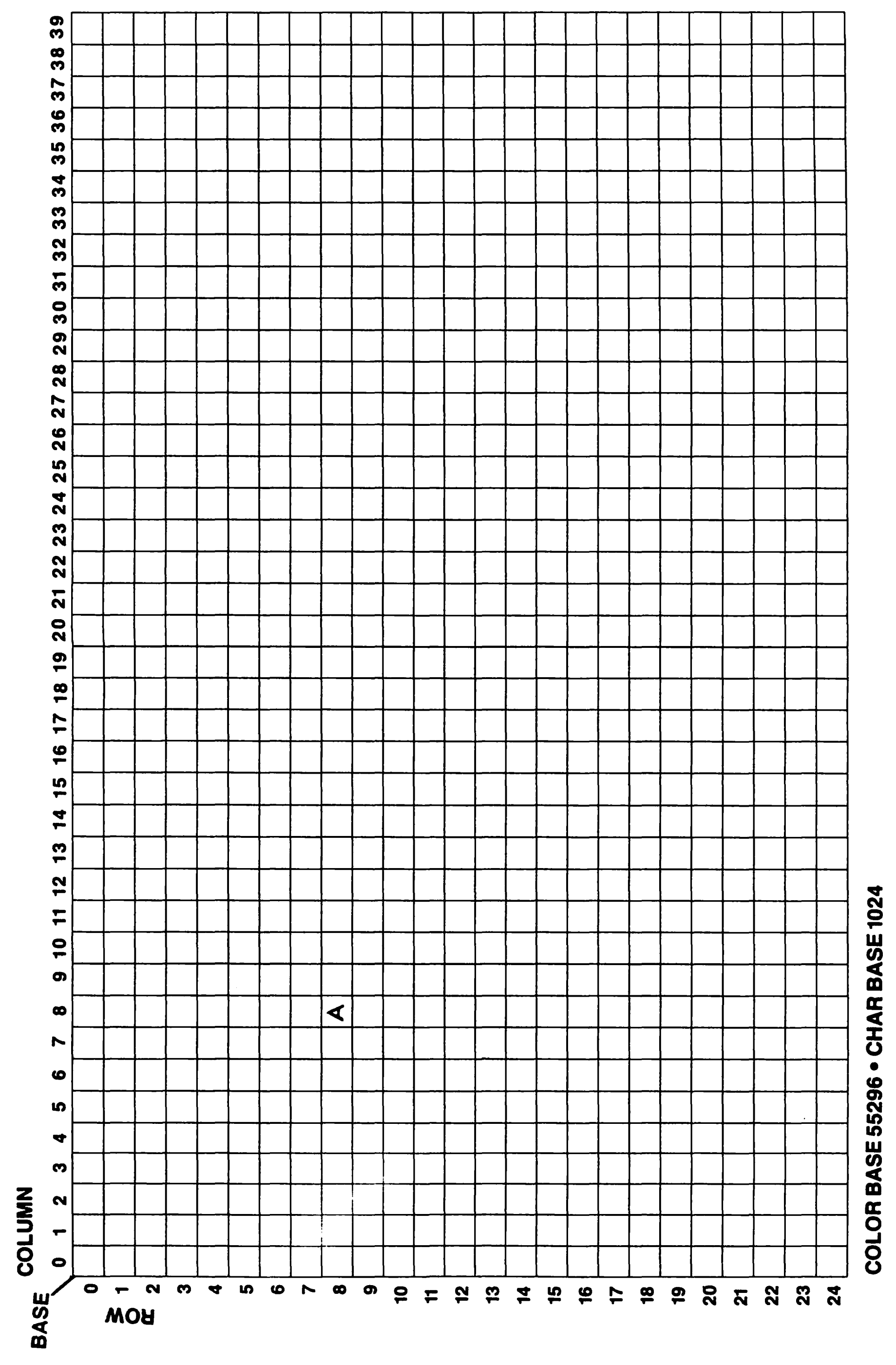
The Screen

SCREEN FORMAT

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 following illustrates the correspondence between the position on the screen and the memory location POKEd.



COLUMN
BASE
ROW
A
COLOR BASE 55296 • CHAR BASE 1024

By adding the column and row values from the chart, then adding this sum to the starting address of the RAM concerned (1024 for characters or 55296 for color), it is easy to position your character and color anywhere on the screen. For example, to put a letter "A" in red at the position marked "A":

```
Row 7 x 40 characters = 280
           + Column 8 = 288
POKE 1024 + 288,1(A)
POKE 55296 + 288,2 (red)
```

The color information is also stored in RAM and starts at 55296. This area holds the number of the color to be displayed. By POKEing the corresponding location in the color RAM, you can change the color of whatever character is displayed.

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

SAVING SCREENS

Consider for a moment that you have formed a complicated display either by POKEing the screen RAM or by PRINTing the information. Chances are that you will want to use this display in various parts of your program. Instead of having to repeat the original process because the screen information is stored in RAM, 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 following programs will move the entire contents of both the screen and the color RAM 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; each screen requiring 1000 bytes for characters and 1000 bytes for color information.

This program 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 color 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 color 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.

MACHINE CODE SCREEN SAVING

The way around the slow data 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 previous programs, only faster.

Quite simply, the FOR-NEXT loop reads from the DATA statements the machine code program and POKEs it into memory. When you wish to save the screen, simply GOSUB10000; 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)<>8THEN GOSUB50000
100 REM-------------------------------
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 takes some time to POKE the machine code in, this can be done and then forgotten.

```
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 **
```

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 located the information in the same place.

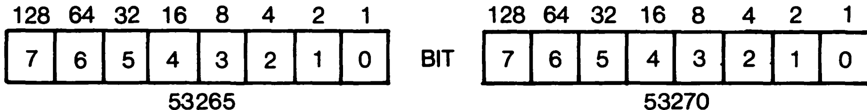
SCREEN SCROLLING

The Commodore 64 will also support *scrolling*. This enables the text or graphics on the display to be moved smoothly in a horizontal or vertical direction.

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

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). As 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 only this bit must be affected.



VIC II control registers

PUT CONTROL REGISTERS

	Register 53265	Register 53270
Bit 0,1,2,	Y scroll pos	X scroll pos
Bit 3	24 or 25 row display, 1 = 25, 0 = 24	38 or 40 columns 1 = 40 columns 0 = 38 columns
Bit 4	blank screen to border color, 1 = on, 0 = off	multicolor mode 1 = on, 0 = off
Bit 5	bit map flag 1 = on, 0 = off	must be set to 0 VIC reset
Bit 6	extended color 1 = on, 0 = off	unused
Bit 7	bit 8 of raster location	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, therefore, 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 a number between 0 and 7 into the registers. This corresponds to 0 (far left or top), and 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.

When the program is run you will see 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.

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

```
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"[CLR]---------- X SCROLL EXAMPLE ------"
130 POKE53270,PEEK(53270)AND247
135 FORN=0TO39
140 FORX=0TO70
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
```

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 that occurs is due to Basic not being fast enough to reset the position to 0. If this whole program was in machine code, the motion would be smoother.

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 be summarized as follows:

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 COLOR MODE

Another mode under which the screen can operate is the EXTENDED COLOR mode. Under normal circumstances, the character is displayed in the character color POKEd or PRINTed against the background color of the screen. In extended color mode, however, you may select the color on which the character is displayed.

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

The mode is controlled 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 color mode:

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

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

SCREEN CODE	BIT 7	BIT 6	COLOR	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 color registers control the background color of the character, and they are the same as the values used when POKEing color to the screen.

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

```
10 REM ** EXTENDED COLOR MODE **
100 PRINT"[CLR]":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"[10 DOWN RED] STANDARD MODE"
150 FORT=1TO5000:NEXT:REM ** WAIT A WHILE
155 PRINT"[HOME 11 DOWN RED] EXTENDED COLOR MODE"
160 POKE53282,4
165 POKE53283,5
170 POKE53284,7
175 REM ** SET BACKGROUND COLOR
180 POKE53265,(PEEK(53265)OR64)
185 REM ** ENTER EXTENDED COLOR MODE
```

When you run the program, 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 color mode.

The first 64 characters will remain unchanged as the register 0 is also the screen color register. The next 64 characters will be displayed against background color 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 color 2 (held in location 53283) and the last 64 characters will be displayed against background color 3 (location 53284).

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

HIGH RESOLUTION BIT MAPPING

By now, you know 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 preferable 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 in addition to 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 in which 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; each row requires 320/8 or 40 bytes. There are 25 x 8 (200) rows of pixels down, resulting in 200 x 40, (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. The map will require 8000 bytes. The most common position is to start the screen at 8192. This will leave about 6K for the Basic program.

Although 6K may not sound like very much, most of the techniques used are simple repetitive routines that do not occupy much space. In any event, machine code will have to be used if graphics are to be displayed with any speed. Machine code programs 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.

For example:

```
POKE51,0:POKE55,0:POKE52,32:POKE56,32
```

This will force Basic to end at 8191 and 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
```

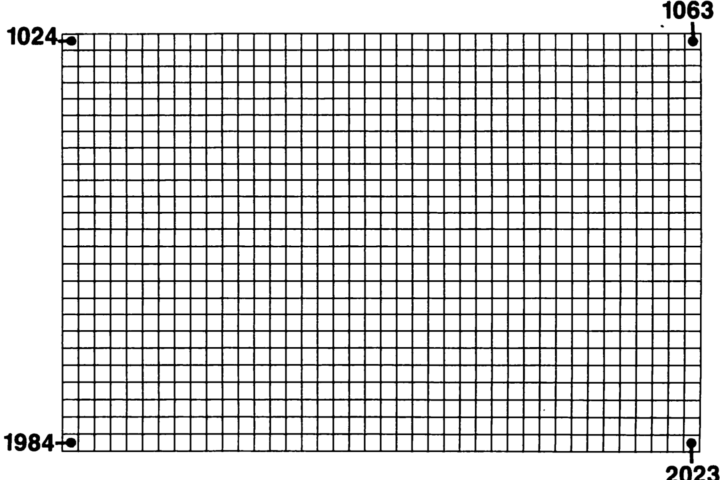
If you enter the program below and RUN it, you will see the screen fill with absolute "garbage." This is because the bit map area is full of random information. To clear the screen, you will need to eliminate the "garbage." The lines below will do this.

```
10 REM ** ENTER BIT MAP **
15 POKE53272,(PEEK(53272)OR8)
20 POKE51,0:POKE55,0:POKE52,32:POKE56,32
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 color that look like ribbons. This is because the area that was used to hold the character information (1024 to 2023) is used to hold the color data in bit map mode.

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

PIXEL				SCREEN				
0	0	0	0	0	1	1	1	1024 = 7
0 = BLACK				7 = YELLOW				



The first four bits (0 to 3) hold the color of bits set to 0 in the map area. The last four bits (4 to 7) hold the color 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 left-most bits will be 0000 (black), value = 16 x 0 = 0
The right-most bits will be 0111 (yellow), value = 7
Put together, it looks like this: 00000111 which is 7 decimal.

This list will help you to set the color to the required values.

PIXEL (SET TO 1)	COLOR (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 15 = 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.

An alternative method is to select the color code for the bits set and multiply by 16, then add the code for the screen color.

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 from the following listing to your program, you will clear the screen to yellow, and any points plotted will be in black.

The complete initialization for bit mapping would now look like this:

```
15 POKE53272,(PEEK(53272)OR8)
20 POKE51,0:POKE55,0:POKE52,32:POKE56,32
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 the same thing is shown below. In order to set the color combinations you require, POKE the calculated value into 247 and call the routine with SYS49152. It performs exactly the same functions as the Basic program, but in less than a second.

```
10 REM ********************************
15 REM * M CODE BIT MAP ENABLE & CLEAR *
20 REM * POKE 247,C TO SET COLOR       *
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 DATA0,165,247,145,180,200,196,170,208
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
```

Once the machine code has been loaded, you can clear the graphics screen by POKEing the color combination you require into 247 and calling 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.

X-Y COORDINATE SYSTEM

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 the need to 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 color (black in the example).

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.

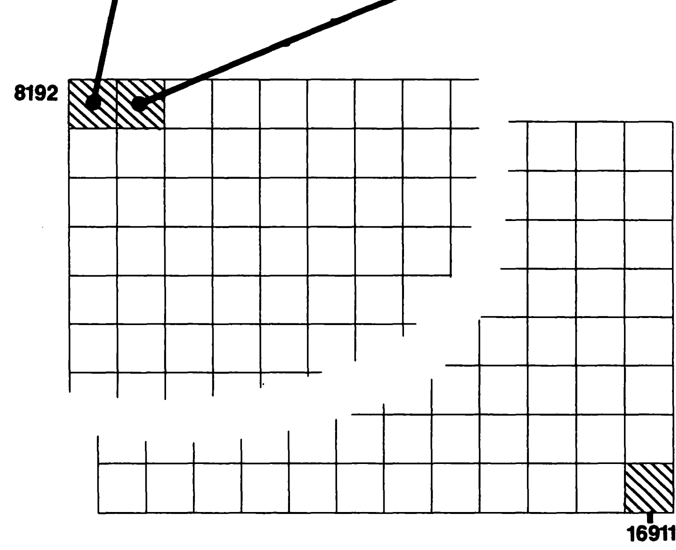
The manner in which the mapping occurs does not lend itself to a direct method of coordinate specification. In order to select whether a particular bit (or pixel on the screen) will be on or off, you need to calculate not only the position on the screen but the location it is controlled by, and finally,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

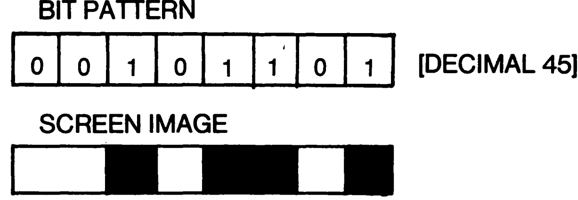
Rather than leave you to struggle with this, a small subroutine has been written that will enable you to enter the X-Y coordinates of a particular pixel and switch it on.

```
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
```

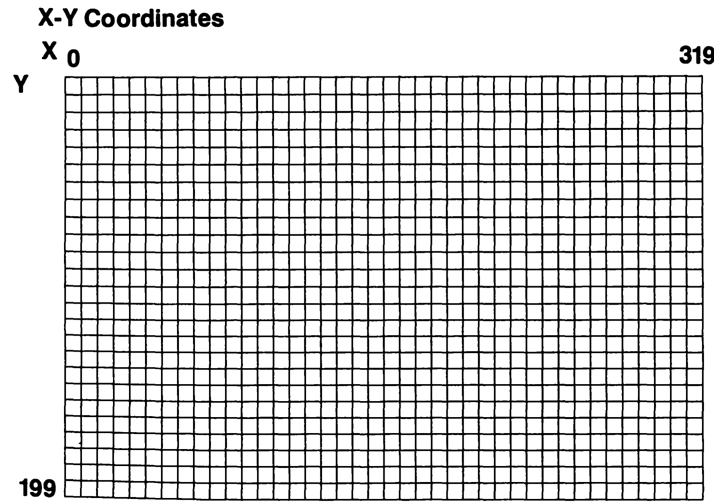
8192 16911

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





If this routine is incorporated in one of the previous initialization programs, 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.



SHAPE PLOTTING

The next program 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)<>8THENGOSUB50000
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 POKEd in. If it has, then the program passes on to the next line. If the routine is not loaded, it will GOSUB 50000 and POKE it.

Line 55 sets the colors 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 and curves.

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
```

(continued)

```
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 show the lines that control the plotting routine. These should be placed in the previous program, beginning at line 100.

```
10 REM *********************************
15 REM **          HIRES CIRCLE        **
20 REM *********************************
25 REM
30 REM
100 XC=160:YC=100:R=80:XO=1:YO=0.S:A=2*
[SHIFT UP ARROW]
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*
[SHIFT UP ARROW]
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 are created.

```
10 REM ********************************
15 REM **            PATTERN         **
20 REM ********************************
25 REM
30 REM
100 XC=160:YC=95:R=40
105 FORO_S=.2TO2STEP.2
110 XO=OS
115 YO=2-OS
120 FORV=0TO2*[SHIFT UP ARROW]STEP.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
```

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.

MULTICOLOR BIT MAPPED GRAPHICS

Multicolor 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 colors to be represented. Although it is possible to select the background color and pixel color for each 8 x 8 block, color changes within that area are not possible.

In multicolor graphics, each pixel is represented by two bits of memory. This allows the use of up to four independent colors for each pixel. The trade-off is that each pixel on the screen appears twice as wide because it is mapped from two bits. In other words, the horizontal resolution is limited to 160 pixels. The vertical resolution of 200 pixels remains unchanged.

In order to select multicolor mode, it is necessary to set the multicolor

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 the bit map mode.

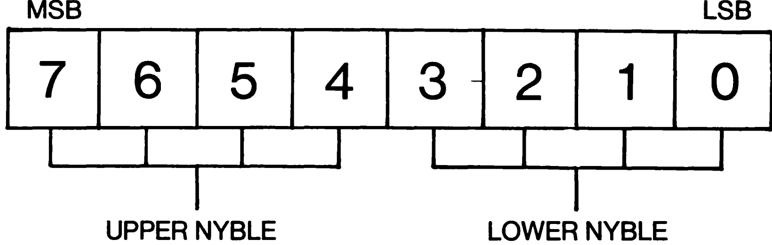
To disable the bit concerned:

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

The correspondence between the bit pairs and the pixel color is shown here:

BITS	COLORS	LOCATION
00	SCREEN COLOR	53281
01	UPPER SCREEN NIBBLE	1024-2023
10	LOWER SCREEN NIBBLE	1024-2023
11	COLOR RAM	55296-56319

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



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

00 color as screen color, i.e. transparent
01 upper screen nibble: is selected by multiplying the normal color code by 16
10 lower screen nibble: the normal color code for that color added to the 01 color value and POKEd to the screen area RAM
11 color RAM color code: POKEd into the color RAM

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

The following 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 color combination for each of the 8 x 8 squares.

```
10 REM *********************************
15 REM **    MULTICOLOR BIT MAPPING    **
20 REM *********************************
16 REM
18 REM
20 POKE51,0:POKE52,32
25 POKE55,0:POKE56,32
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 color being the same as the screen color, 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 multicolor screen, you have to specify in which color the point is to be plotted in addition to specifying the X-Y coordinates.

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

The color of each pixel is determined by the value of the variable CO in Line 110.

```
10 REM ********************************
15 REM *  MULTICOLOR 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=S*((Y/8)-A)
1010 C=INT(XC/8)
1011 PI=S*((XC/8)-C:PI=6-PI
```

(continued)

```
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 DATA249,230,181,201,7,240,6
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 method used to generate multicolor graphics is similar to the one previously described. The main differences are that in multicolor mode, you have to specify the color the pixel will be displayed in, and the X resolution is halved.

Finally, remember to make sure that the colors chosen will be visible when they are displayed.

MOVING THE SCREEN

All the graphics facilities on the Commodore 64 are controlled by the VIC II chip. This is an extremely powerful microcircuit but 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 programs. This is because the VIC II chip can only "see" (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 1000 bytes for the

screen character memory. It is only when using bit mapped 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 position 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 POKE51,0:POKE52,128
```

(continued)

```
105 POKE55,0:POKE56,128
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
```

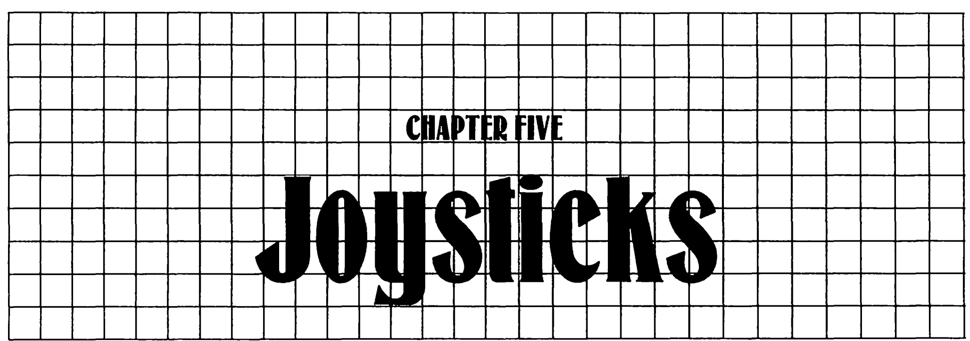
The new locations will be:

CONTENT	PREVIOUS POSITION	NEW POSITION
Screen memory	1024-2023	35840-36839
Character data	RAM	32768-34815
Sprite data area	Any Place < 16384	34816-35839
Sprite pointers	2040-2047	36856-36863
The color RAM	55296-56295	55296-56295

The program 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 programs 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 programs. This, together with the storage for graphic data, will enable the programmer to write far more complex programs than would be possible with the standard configuration.



Port Connections

The Commodore 64 allows the use of two joysticks. Each joystick is connected to an 8 bit *port*. The port can be viewed as a location in memory. To read a joystick value, or 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 here:

```
BIT NUMBER 7 6 5 4 3 2 1 0
--------------------------
           0 0 0 0 0 0 0 0
           X X X f r l d u



PORT 1 connections
location 56321
X = unused (set to 1)
f = fire
r = right
l = left
d = down
u = up
```

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

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

For example, to read a joystick in port 1:

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

If you run the program 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 logic 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 JOYTICK 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$= "FIRE"
1026 REM IF BIT 4 SET JOYSTICK IS FIRING
1030 IF JS = 0 THEN A$="CENTER"
1031 REM IF NONE SET JOYSTICK IS CENTERED
1035 PRINT "[CLR]JOYSTICK POSITION =";A$
1040 GOTO1000
```

This program has one big flaw: it will not respond to combined positions, for example UP and FIRE.

This next routine will overcome this.

```
10 REM ** JOYSTICK COMBINED **
100 A=255-PEEK(56321)
110 IFAAND1THENA$="UP"
120 IFAAND2THENA$=A$+"DOWN"
130 IFAAND4THENA$=A$+"LEFT"
140 IFAAND8THENA$=A$+"RIGHT"
150 IFAAND16THENA$=A$+"FIRE"
160 IFA$=""THENA$="CENTER"
170 PRINT"[CLR] JOY 1 ";A$:A$=""
180 GOTO100
```

Enter the above program to make sure the screen displays the expected position.

In order to facilitate the use of the information on the joystick position, assign a variable to each position and test each one individually in the main program. 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"[CLR] 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 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 program is shown below.

```
10 REM ** USER JOY TWO **
100 GOSUB1000
105 PRINT"[CLR] 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
```

USING TWO JOYSTICKS

The following program combines the two shorter programs to read both joysticks. This time, instead of coding decisions in the program for each joystick position, we read all the information about the joystick directions and memory locations in the DATA line.

```
10 REM ** USER BOTH JOYS **
20 PRINT"[CLR]";
30 FOR J= 0 TO 4:READ D$(J),P(J):NEXT J
40 READ M(1),M(2),J(1),J(2)
50 DATA UP,1,DN,2,LF,4,RT,8,FR,16,255,127,56321,
56320
60 PRINT"[HOME]";
100 FOR JS= 1 TO 2
110 GOSUB1000:PRINT"JOYSTICK #"JS
120 FOR J=0 TO 4
130 PRINT D$(J)"=";V(J)
140 NEXT J:PRINT:NEXT JS
150 GOTO 60
1000 A=M(JS)-PEEK(J(JS))
1010 FORJ=0TO4
1020 V(J)=0:IF(A AND P(J))THEN V(J)=1
1030 NEXTJ:RETURN
```

GRAPHICS

One very powerful combination is the ability to use joysticks in graphic design. Although programs that exploit the possibilities to the greatest extent are complex and lengthy, it is possible to demonstrate the capabilities with the following, simple program.

```
10 REM ********************************
15 REM * JOYSTICK BIT MAPPED GRAPHIC *
20 REM ********************************
25 REM
30 REM
50 POKE51,0:POKE52,32:POKE55,0:POKE56,32
55 IFPEEK(49152)<>8THENGOSUB5000
60 POKE247,80:SYS49152
65 X=160:Y=100:GOSUB1000
100 A=255-PEEK(56321):XC=0:YC=0
105 IFAAND1THENYC=1
110 IFAAND2THENYC=+1
115 IFAAND4THENXC=-1
120 IFAAND8THENXC=+1
```

(continued)

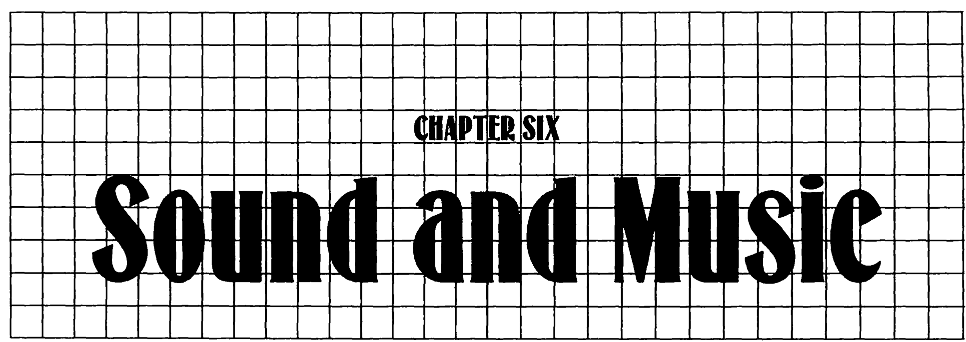
```
125 IFAAND16THENFB=1
130 X=X+XC:Y=Y+YC
135 IF X<1THENX=0:GOTO100
140 IFX>319THENX=319:GOTO100
145 IFY<1THENY=1:GOTO100
150 IFY>199THENY=199:GOTO100
155 GOSUB1000
160 IFFB=1THENFB=0:GOTO200
165 GOTO100
200 XB=X-XC:YB=Y-YC
205 A=INT(YB/8):B=8*((YB/8)-A)
210 C=INT(XB/8):P=8*((XB/8)-C):P=7-P
215 BY=((A*320)+(C*8)+B)8192
220 POKEBY*PEEK*BY)AND(255-2↑P)
225 GOTO100
1000 A=INT(Y/8*((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 DATA133,180,133,170,169,4,133,181,169
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 DTAT170,76,64,192
50000 FORN=0TO95
50005 READA
50010 POKEN+49152,A
50015 NEXT
50020 RETURN
```

Once again, the DATA lines contain our familiar subroutine. You do not have to re-enter them again.

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 program is straightforward 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 color of the screen and pixel plot may be modified in a similar manner to that already described (see bit mapping) by changing the number POKEd into location 247, line 50.



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 deals with ways to generate special sound effects and to 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 steps involved in making sound can 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 allows the use of one oscillator to control another which widens the range of effects available.

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

VOLUME

The volume or loudness is controlled by a register in the SID chip. This register 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 (maximum) into the register.

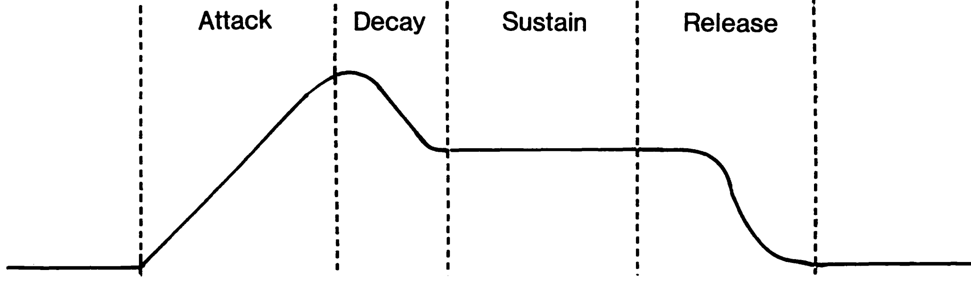
POKE54296,15 will set the volume for all sound output to the maximum level.

POKE54296,0 will turn the volume off.

In all the examples in this book, the volume is set to 15. You will be able to control the actual volume by using the control on your monitor.

ADSR

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. Sustain in the SID chip, however, is defined slightly

differently. Rather than a time of constant value, it is a level (how loud) of volume. 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, has a small decay, a long sustain and a long release.

The SID chip will allow you to modify the overall envelope using 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	ATTACH/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 part, bits 3-0.

In the attack-decay byte, the attack is controlled by the first part (bits 7-4) and the release by the second part (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; 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	2000 (3 S)	9000 (9 S)
14	5000 (5 S)	15000 (15 S)
14	8000 (8 S)	24000 (24 S)

*mS = milliseconds (1 millisecond = 1000th second)

The sustain value sets the level of the volume for the duration of the note. 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 the attack and sustain values must be multiplied by 16 and added to the decay-release figures.

For example, here is a set of parameters with assigned values:

OSCILLATOR 1

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

Combined attack/decay = 7 x 16 + 3 = 115
Combined sustain/release = 15 x 16 + 6 = 246
POKE 54277,115
POKE 54278,246

The sustain value only controls the maximum volume, not the duration of the note. The usual method of controlling the duration 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 waveform)
FOR T = 1 TO 500:NEXT (duration of note approximately 1/2 second).
POKE54276,16 (turn off voice 1)

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

The lifetime of a single note can be summarized as follows:

- 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.

PITCH

The pitch or note of the oscillator is controlled by two registers for each voice. This allows an effective range of 0 to 65535 different notes, allowing 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 as follows:

HIGH BYTE = INT(VALUE/256)
LOW BYTE = VALUE — INT(VALUE/256)*256

where VALUE is the value of the note required.

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

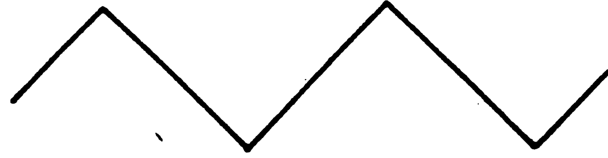
NOTE	VALUE	HIGH	LOW
C-4	4291	16	195
C#	4547	17	195
D-4	4817	18	209
D#	5103	19	239
E-4	5407	21	31
F-4	5728	22	96
F#	6069	23	181
G-4	6430	25	30
G#	6812	26	156
A-4	7217	28	49
A#	7647	29	223
B-4	8101	31	165
C-5	8583	33	135
C#	9094	35	134
D-5	9634	37	162
D#	10207	39	223
E-5	10814	42	62
F-5	11457	44	103
F#	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

WAVEFORM

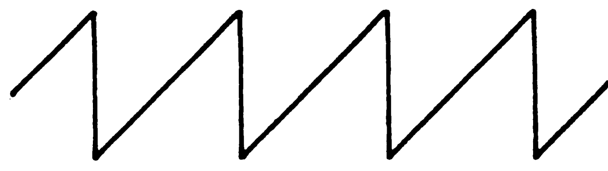
Each oscillator is capable of producing four different waveforms, one at any given moment.

They are:

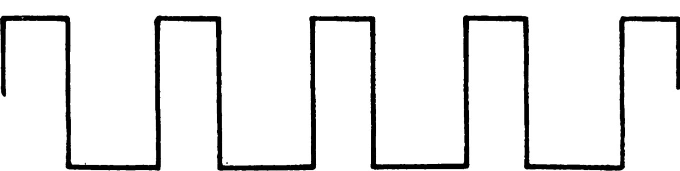
- Triangle



- Sawtooth



- Pulse



- 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 three registers, once for each oscillator. Their locations are:

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
POKE 54276,129 sets osc. 1 to NOISE

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

OSCILLATOR 1
POKE54276,16 turn off TRIANGLE
POKE54276,32 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.

The pulse waveform has variable width which is set by the user. The registers that control the pulse width are:

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

With three registers, the value is 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. For example:

A VALUE of 2048 will give a 50% duty cycle — 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 following formula calculates the numbers to be POKEd:

HIGH = INT (USE/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 modifications, we will examine a few of the previous concepts in more detail.

Any program using sound must follow certain principles in order for it to work. These principles are:

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 value 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 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 programs use sound to enhance their appeal. The sound effects most often used 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 program, it would be sensible to structure the programs as subroutines that may be called by various sections of the main program.

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. They should be placed near the start of the main program to ensure that 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
```

Line 5000 sets the attack-decay.

Line 5005 sets the waveform to NOISE.

Line 5010 sets the sustain-release.

Line 5015 sets the High byte for the frequency.

Line 5020 sets the Low byte for the frequency.

Line 5025 turns the waveform off.

Line 5030 returns to the main program.

An interesting feature of this routine is that the duration of the sound is controlled by the RELEASE setting. This is to allow the effect to continue without interrupting any other action, such as graphics, that may be occurring.

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

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

```
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
```

Line 5000 sets the attack-decay.

Line 5005 sets the sustain-release.

Line 5010 sets the low byte for the frequency.

Line 5015 turns on the noise waveform for osc. 1

Line 5020 starts a loop to modify the frequency of osc. 1

Line 5025 changes the frequency of the noise.

Line 5030 increments the loop.

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

```
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 88 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
```

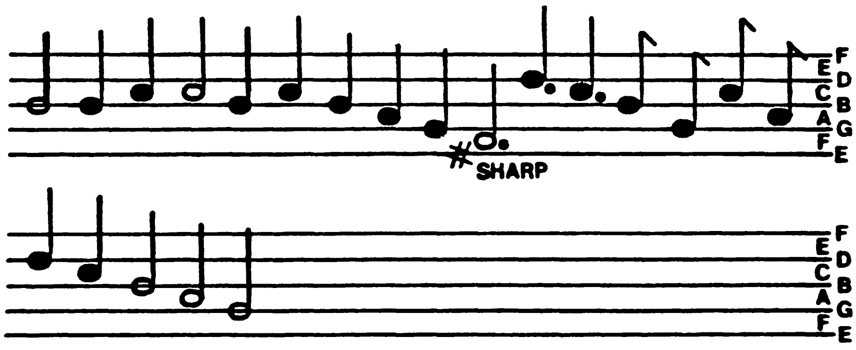
MUSIC

The sound generators in the Commodore 64 may also be used to play music. The fundamental principles of sound formation have already been explained 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 staff and the duration by the appearance of the notes.

Here is an example:



The duration of notes are as follows:

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 1.5 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:

𝄿 1/16 𝄾 1/8 𝄽 1/4 𝄼 1/2 𝄻 1

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.

When a rest occurs, the VALUE (pitch) of the note is set to 0 and the DURATION value is set according to the previous table.

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:NEXTN
100 POKE54296,15   :REM ** VOLUME
105 POKE54277,155  :REM ** ATTACK/DECAY
110 POKE54278,246  :REM ** SUST/RELEASE
115 READF,D:PRINTF,D
```

(continued)

```
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 DATA81081,8,8101,4,8583,4
505 DATA8583,8,81081,4,8583,4
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
```

Line 50 clears the SID chip registers.

Line 100 sets the volume to maximum.

Line 105 sets the attack-decay.

Line 110 sets the sustain-release.

Line 115 reads the values for the note and duration into variables F (frequency) and D (duration).

Line 120 checks to see if the tune has been played.

Line 125 calculates the high byte for the note.

Line 130 calculates the low byte.

Line 135 POKEs the high and low byte into voice 1 frequency registers.

Line 140 turns the sound on.

Line 145 uses the delay value to determine the duration of the note.

Line 150 turns the note off.

Line 155 loops back to play the next note.

Lines 500-530 contain the data for the note and duration.

Line 540 holds 2 zeros to indicate that all the music has been played.

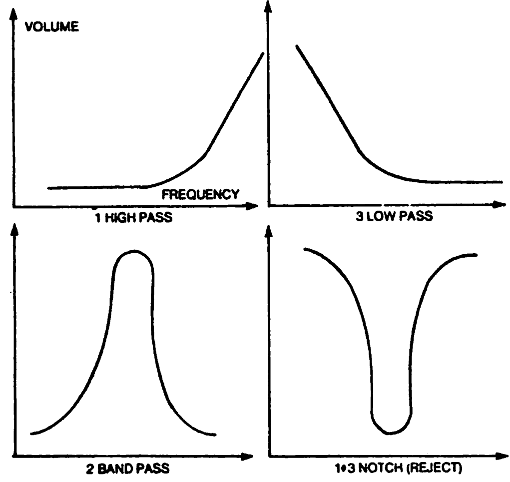
By using different settings for the generation of the sound, a wide variety of pleasing tones may be produced. For example, the changes below will cause the sound to imitate a guitar.

```
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
```

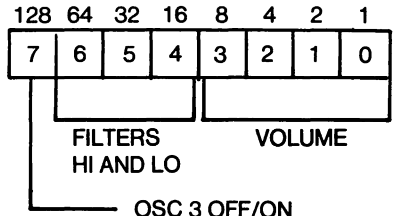
FILTERING

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 basic type of filters (HIGH, BAND and LOW) and their operation may be combined to form others.



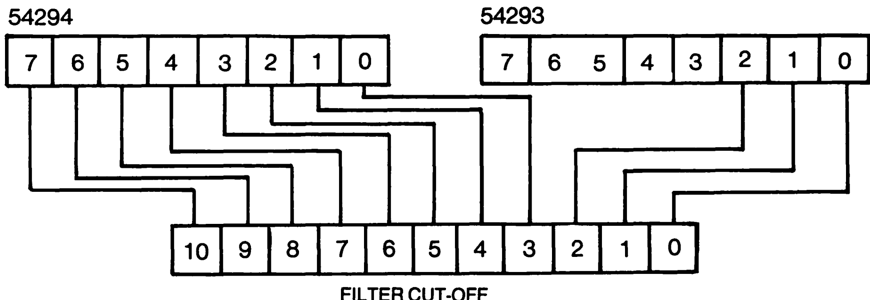
Each filter is controlled by a bit in the same register (54296) that controls the overall volume.



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 you can see, 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.

RESONANCE

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

RING MODULATION

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 synchronize 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

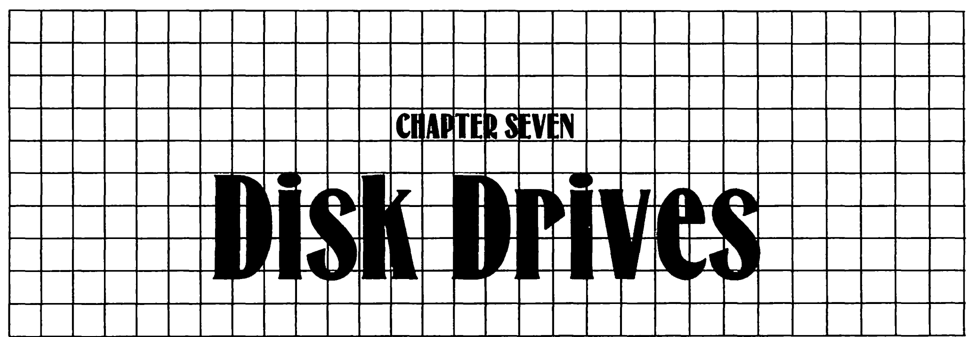
OSC 3 with OSC 2
Add 4 to the WAVEFORM VALUE and POKE it into 54290

SYNC OSC 1 with 3
Add 2 to the waveform and ring modulation figure and POKE it 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 waveform and ring modulation figure and POKE it into 54290

It is not possible 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. We hope that this brief introduction to the SID chip will provide enough raw material to allow the user to experiment further: for example, try altering the pitch of an oscillator while it is on and dynamically changing the filtering.



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 needed 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.

DEVICE NUMBERS

The disk drive is normally accessed as device 8.

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

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

When using more than one disk drive, it is necessary to change 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 "device 8" and the second "device 9".

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 repairman.

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 program here will allow you to change the device number.

```
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 program 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 drive switched on, the new device number is used to access that disk. For example:

SAVE"DEVICE CHANGE",9

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

LOAD"ACCOUNTS",8

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

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

DISK ERRORS

Whenever the disk controller detects an error condition, for example, 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 program, it is a good idea to keep track of any errors by checking the error status 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$,DT,DS
50005 IFEN<20THENRETURN
50020 PRINT"DISK ERROR"
50025 PRINT"ERROR NUMBER"EN
50030 PRINT"ERROR TYPE"M$
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"THENPRINT:RETURN
50065 CLOSE1:END
```

The error channel is OPENed in line 100; this should be placed at the front of the program 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 program.

Lines 50005 to 50040 display the exact nature of the error on the screen and lines 50045 to 50065 allow you to end the program or continue. Bear in mind that you have to use your judgment about whether or not to continue. For instance, in a "disk full situation", it doesn't make sense to continue.

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

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 program, or you may wish to set up several different screens in various places in a program. 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 a routine. Note that this is coded for device 9. If your disk is device 8, change "9" to "8" in line 50005 in this program and in the next program.

```
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;CHR$(13);
50020 A=PEEK(55296+N):PRINT#2,A;CHR$(13);
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. Note that a carriage return (CHR$(13) is forced between the values and that a linefeed character is suppressed, as it can cause problems in many disk systems during subsequent INPUT.

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
```

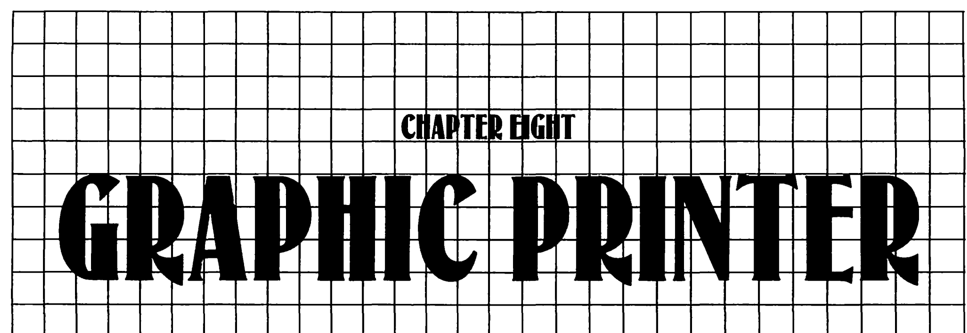
```
50020 INPUT#2,A:POKE55296+N,A
50025 NEXT
50030 CLOSE2
50035 RETURN
```

DISK MAINTENANCE

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

The Commodore 64 does not have a built in language to perform disk maintenance. There is, however, a program on the TEST-DEMO disk called "DOS WEDGE" (or something similar). You will find this program one that will be very useful since it simplifies working with the disk. LOAD and RUN it.

If you must perform any disk maintenance commands from a program (as we did in the DISK ERROR subroutine), use PRINTING to and INPUTTING from channel 15. The disk manual describes the necessary procedures.



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

NAME AND ADDRESS LABELS

The program that follows will print name and address on self-adhesive labels, 1.5 by 3.5 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 program 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 ZIP code.

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

After the printing has finished, you may print another copy of the label or enter new information. A sample label is printed below.

```
MICROBOOKS
443 MILLBROOK ROAD
SOUTHAMPTON
HAMPSHIRE
SO1 OHX
```

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 it is difficult to remember the required code.

One 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
```

(continued)

```
120 STNDRD$=CHR$(15):REM * NORMAL MODE
125 SPC$=CHR$(16): REM * POSITION
130 DT$=CHR$(27): REM * DOT POSITION
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

At times, you may want to produce "hard copy" of the screen display. The following program will print the entire contents of the screen and it also checks for upper/lower case or upper case/graphics to ensure that the copy is accurate. The routine will not copy reverse characters that follow a quote correctly; also it 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
050025
50015 S$=CHR$(145)
50020 OPEN4,4:PRINT#4,S$
50025 SB=1024
50030 FORP=SBTO2023
50035 C=PEEK(P) :C$=""
50040 IF(P-SB)/40=INT((P-SB)/40)THENPRINT
#4,CHR$(13)+CHR$(15);
50045 IFC>128THENC=C-128:C$=CHR$(18)
50050 IFC<320ORC>95THENC=C+64:GOTO50060
50055 IFC>63ANDC<96THENC=C+128
50060 C$=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
```

Computers $14.95

Add new dimensions of color, sound, and graphics to your BASIC programs.

PROGRAMMING TIPS FOR THE COMMODORE 64™

DAVID HIGHMORE
LIZ PAGE

If you have a good grasp of BASIC and are ready to master the special programming features and techniques of the Commodore 64, this intermediate guide is for you. It's designed to help you build more sophisticated programming skills as you create colorful graphics, animation, music and sound effects. You'll learn quick techniques for performing more advanced operations, including:

- Drawing and moving "Sprite" graphics
- Multicolor graphics
- High resolution bit mapping
- Programming with a disk drive
- Using a graphic printer

In addition, *Programming Tips for the Commodore 64* reviews the fundamental aspects of the Commodore 64 such as its basic statements and function keys. Fully coded routines, annotated and illustrated, help you tackle complex tasks with ease and confidence.

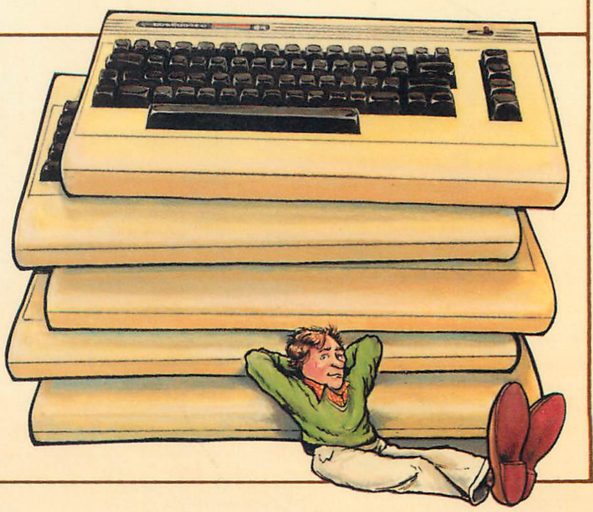
DAVID HIGHMORE is a computer software consultant and Director of Micro Books, Ltd.

LIZ PAGE is a programmer and software developer.

Commodore 64 is a registered trademark of Commodore Electronics, Ltd.

Wiley Press guides have taught more than three million people to use, program, and enjoy microcomputers. Look for them all at your favorite bookshop or computer store.

Jacket Illustration: Dennis Dittrich



WILEY PRESS
a division of JOHN WILEY & SONS, Inc.
605 Third Avenue, New York, N.Y. 10158
New York • Chichester • Brisbane
Toronto • Singapore

ISBN 0 471-81553-5